目錄

RustPrimer

build passing

The Rust primer for beginners.

给初学者的Rust中文教程。

在线阅读地址

点我阅读

也可复制以下链接：

```
https://rustcc.gitbooks.io/rustprimer/content/
```

社区

QQ群

群号：

```
    Rust语言中文社区 144605258 （已满）
    Rust编程语言社区2群 303838735
```

进群必须附带您的*github*地址，否则不予通过

社区

chat: https://chat.rust-china.org/ (使用github验证登录)

blog: https://rust-china.org/

wiki: https://wiki.rust-china.org/

版权规定

GitBook 生成

直接用：

```
gitbook serve
```

即可。

ChangeLog

1. 2016年3月31日，初稿完成。发布 v1.0 版。
2. 2016年5月2日，完成 1.1.0 版本。

Rust 是一门系统级编程语言，被设计为保证内存和线程安全，并防止段错误。作为系统级编程语言，它的基本理念是“零开销抽象”。理论上来说，它的速度与 C / C++ 同级。

Rust 可以被归为通用的、多范式、编译型的编程语言，类似 C 或者 C++。与这两门编程语言不同的是，Rust 是线程安全的！

Rust 编程语言的目标是，创建一个高度安全和并发的软件系统。它强调安全性、并发和内存控制。尽管 Rust 借用了 C 和 C++ 的语法，它不允许空指针和悬挂指针，二者是 C 和 C++ 中系统崩溃、内存泄露和不安全代码的根源。

Rust 中有诸如 if else 和循环语句 for 和 while 的通用控制结构。和 C 和 C++ 风格的编程语言一样，代码段放在花括号中。

Rust 使用实现（implementation）、特征（trait）和结构化类型（structured type）而不是类（class）。这点，与基于继承的OO语言 C++, Java 有相当大的差异。而跟 Ocaml, Haskell 这类函数式语言更加接近。

Rust 做到了内存安全而无需 .NET 和 Java 编程语言中实现自动垃圾收集器的开销，这是通过所有权/借用机制、生命周期、以及类型系统来达到的。

下面是一个代码片段的例子，经典的 Hello World 应用：

```
fn main() {
  println!("hello, world");
}
```

影响了 Rust 的流行的编程语言包括 C, C++, C#, Erlang, Haskell, OCaml, Ruby, Scheme 和 Swift 等等。Rust 也影响了 C# 7, Elm, Idris, Swift。

Rust 提供了安装程序，你只需要从官网下载并在相应的操作系统上运行安装程序。安装程序支持 Windows、Mac 和 Linux（通过脚本）上的32位和64位 CPU 体系架构，适用 Apache License 2.0 或者 MIT Licenses。

Rust 运行在以下操作系统上：Linux, OS X, Windows, FreeBSD, Android, iOS。

简单提一下 Rust 的历史。Rust 最早是 Mozilla 雇员 Graydon Hoare 的一个个人项目，从 2009 年开始，得到了 Mozilla 研究院的支助，2010 年项目对外公布。2010～2011 年间实现的自举。从此以后，Rust 经历了巨大的设计变化和反复（历程极其艰辛），终于在 2015 年 5 月 15日发布了 1.0 版。在这个研发过程中，Rust 建

立了一个强大活跃的社区，形成了一整套完善稳定的项目贡献机制（这是真正的可怕之处）。Rust 现在由 Rust 项目开发者社区（https://github.com/rust-lang/rust ）维护。

自 15 年 5 月 1.0 发布以来，涌现了大量优秀的项目（可以 github 上搜索 Rust 查找），大公司也逐渐积极参与 Rust 的应用开发，以及回馈开源社区。

本书（RustPrimer）旨在为中文 Rustaceans 初学者提供一个正确、最新、易懂的中文教程。本书会一直完善跟进，永不停歇。

本书是整个 Rust 中文社区共同努力的结果。其中，参与本书书写及校订的 Rustacean 有（排名不分先后）：

- daogangtang（Mike猫）
- wayslog（猫猫反抗团团长）
- marvin-min
- tiansiyuan
- marvinguo
- ee0703
- fuyingfuying
- qdao
- JohnSmithX
- stormgbs (AX)
- tennix
- anzhihun
- zonyitoo（Elton, e猫）
- 42
- Naupio（N猫）
- F001（失落的神喵）
- wangyu190810
- domty
- MarisaKirisame（帅气可爱魔理沙）
- Liqueur Librazy
- Knight42
- Ryan Kung
- lambdaplus
- doomsplayer
- lucklove

- veekxt
- lk-chen
- RyanKung
- arrowrowe
- marvin-min
- ghKelo
- wy193777
- domty
- xusss
- wangyu190810
- nextzhou
- zhongke
- ryuki
- codeworm96
- anzhihun
- lidashuang
- sceext2
- loggerhead
- twq0076262
- passchaos
- yyrust
- markgeek
- ts25504
- overvenus
- Akagi201
- theJian
- jqs7
- ahjdzx
- chareice
- chenshaobo
- marvinguo
- izgzhen
- ziqin
- peng1999

等。在此，向他们的辛苦工作和无私奉献表示尊敬和感谢！

祝用 Rust 编程愉快！

安装Rust

本章讲解在三大平台 Linux, MacOS, Windows 上分别安装 Rust 的步骤。

Rust for Linux

Rust 支持主流的操作系统，Linux,Mac和 windows。

Rust 为Linux用户提供了两种安装方式：

1、直接下载安装包：

直接下载安装包的话需要检查一下你当前操作系统是64位还是32位，分别下载对应的安装包。

查看操作系统请在终端执行如下命令:

```
uname -a
```

结果如下图所示：

```
[root@cloud ~]# uname -a
Linux cloud 2.6.32-358.el6.x86_64 #1 SMP Fri Feb 22 00:31:26 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
```

如上图所示，如果是 **x86_64** 则证明是64位系统，需要下载64位安装包；

如果是**x86-32**则需要下载32位安装包

下载安装包后解压运行即可。在书写本章时，最新的稳定版本为*1.5.0*，解压： `tar -zxvf rust-1.5.0-x86_64-unknown-linux-gnu.tar.gz`

解压完进入对应的目录： `cd rust-1.5.0-x86_64-unknown-linux-gnu` 执行 `./install.sh`

上述命令执行完成后会打印： **Rust is ready to roll.** 表明安装成功

此时执行: `rustc --version`，你会看到对应的 rust 版本信息,如下图所示：

```
[root@cloud rust-1.5.0-x86_64-unknown-linux-gnu]# rustc --version
rustc 1.5.0 (3d7cd77e4 2015-12-04)
```

2、命令行一键安装：

Rust 提供简单的一键安装，命令如下：

```
curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

打开终端执行如上命令即可。

注意

除了稳定版之外，Rust 还提供了 Beta 和 Nightly 版本，下载地址如下：
https://www.rust-lang.org/zh-CN/other-installers.html

如果你不想安装 Rust 在你的电脑上，但是你还是像尝试一下 rust，那么这里有一个在线的环境：http://play.rust-lang.org/

中国科学技术大学镜像源包含 rust-static，梯子暂时出问题的同学可以尝试从这里下载编译器；除此之外。还有 Crates 源，详见这里的说明。

Rust for Mac OS

Rust 支持主流的操作系统，Linux，Mac 和 windows。

Rust 为 mac 用户提供了两种安装方式：

1、直接下载安装包：

直接下载安装包的话需要检查一下你当前操作系统是64位还是32位，分别下载对应的安装包。 查看操作系统请在终端执行如下命令:

```
uname -a
```

```
✘ ✝ ~/projects/RustPrimer  master±  uname -a
Darwin ML00070039.local 14.5.0 Darwin Kernel Version 14.5.0: Wed Jul 29 02:26:53
 PDT 2015; root:xnu-2782.40.9~1/RELEASE_X86_64 x86_64
```

如上图红色部分所示，如果是 **x86_64** 则证明是64位系统，需要下载64位安装包；如果是**x86-32**则需要下载32位安装包

和安装普通的软件一样，直接运行安装包即可。

在书写本章时，最新的稳定版本为*1.5.0*，

2、命令行一键安装：

Rust 提供简单的一键安装，命令如下：

```
curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

此过程，有可能需要你输入几次密码

你只需打开你的命令行执行如上代码就可以了。（注意，你可能需要一个梯子，否则会遇到一些类似*Could not resolve host: static.rust-lang.org*的错误）

3.验证安装：

如果你完成了上面任意一个步骤，请执行如下命令：

```
rustc --version
```

如果看到如下信息，表明你安装成功：

```
rustc 1.5.0 (3d7cd77e4 2015-12-04)
```

如果提示没有 *rustc* 命令，那么请回顾你是否有某个地方操作不对，请回过头来再看一遍文档。

注意

除了稳定版之外，Rust 还提供了 Beta 和 Nightly 版本，下载地址如下：https://www.rust-lang.org/zh-CN/other-installers.html

如果你不想安装 Rust 在你的电脑上，但是你还是像尝试一下 rust，那么这里有一个在线的环境：http://play.rust-lang.org/

中国科学技术大学镜像源包含 rust-static，梯子暂时出问题的同学可以尝试从这里下载编译器；除此之外，还有 Crates 源，详见这里的说明。

Rust for Windows

Rust 支持主流的操作系统，Linux,Mac和 Windows。

Rust在Windows上的安装和你在windows上安装其它软件一样。

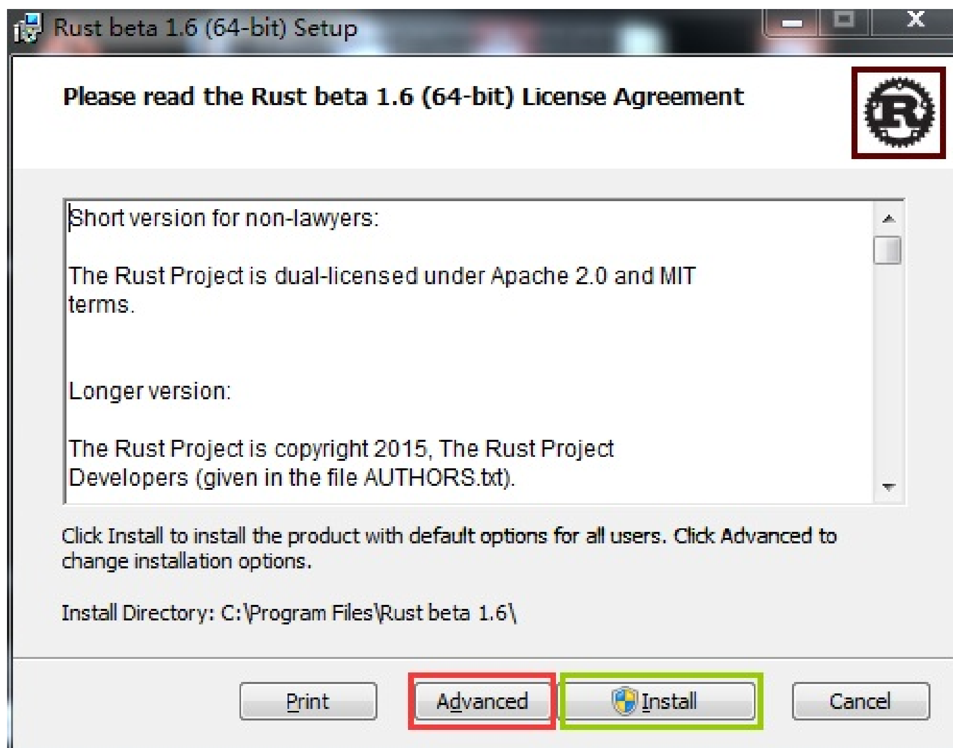
1、下载安装包：

下载地址

Rust提供了多个版本和多个平台的安装包，下载对应的即可，此处我们以1.6.0的稳定版为例。

2、安装：

双击下载到的安装包，如下图所示：



默认，rust将安装到所有用户下，选择“Advanced”，可以指定安装用户和安装路径。然后点击"install"等待几分钟即可（中间可能会有安全提示，点击允许即可，如果你装了360之类的，需要小心360阻止写入注册表）。

3.验证安装：

安装完成后，运行windows命令行，然后输入:

```
rustc --version
```

看到 以 **rustc 1.6.0** 开头，说明你安装成功了。

注意

中国科学技术大学镜像源包含 rust-static，梯子暂时出问题的同学可以尝试从这里下载编译器；除此之外。还有 Crates 源，详见这里的说明。

Rust 版本管理工具: rustup

rustup 是rust官方的版本管理工具。应当作为安装 Rust 的首选。

项目主页是: https://github.com/rust-lang-nursery/rustup.rs

Features

- 管理安装多个官方版本的 Rust 二进制程序。
- 配置基于目录的 Rust 工具链。
- 安装和更新来自 Rust 的发布通道: nightly, beta 和 stable。
- 接收来自发布通道更新的通知。
- 从官方安装历史版本的 nightly 工具链。
- 通过指定 stable 版本来安装。
- 安装额外的 std 用于交叉编译。
- 安装自定义的工具链。
- 独立每个安装的 Cargo metadata。
- 校验下载的 hash 值。
- 校验签名 (如果 GPG 存在)。
- 断点续传。
- 只依赖 bash, curl 和常见 unix 工具。
- 支持 Linux, OS X, Windows(via MSYS2)。

安装

Windows

在rustup的主页下载并运行rustup-init.exe,并按照提示选择选项。

```
Welcome to Rust!

This will download and install the official compiler for the Rus
t programming
language, and its package manager, Cargo.

It will add the cargo, rustc, rustup and other commands to Cargo
's bin
directory, located at:

  C:\Users\Liqueur Librazy\.cargo\bin

This path will then be added to your PATH environment variable b
y modifying the
HKEY_CURRENT_USER/Environment/PATH registry key.

You can uninstall at any time with rustup self uninstall and the
se changes will
be reverted.

Current installation options:

   default host triple: x86_64-pc-windows-msvc
     default toolchain: stable
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
```

三个选项分别是

1) 开始安装（默认选项） 2) 自定义安装 3) 取消

其中自定义安装可以更改默认架构与工具链、是否添加 PATH。例如想要选择 nightly 工具链可以进行以下自定义

```
I'm going to ask you the value of each these installation option
s.
You may simply press the Enter key to leave unchanged.

Default host triple?


Default toolchain? (stable/beta/nightly)
nightly

Modify PATH variable? (y/n)
```

设置完毕后，选择 1 以开始安装。

Linux & macOS

运行以下命令

```
curl https://sh.rustup.rs -sSf | sh
```

这个命令将会编译和安装 rustup, 安装过程中可能会提示你输入 sudo 的密码。 然后, 他会下载和安装 stable 版本的工具链, 当执行 rustc, rustdoc 和 cargo 时, 将会配置他为默认工具链。

`Unix` 上安装后工具链会被安装到 `$HOME/.cargo/bin` 目录。

`.cargo/bin` 目录会被添加到系统的 `$PATH` 环境变量,重新登录后即可使用 `rustc` ，`cargo` 等命令。

卸载

```
rustup self uninstall
```

用法

安装后会得到一个 rustup 命令, 多使用命令自带的帮助提示, 可以快速定位你需要功能。

帮助

运行 `rustup -h` 你将会得到如下提示:

```
❯ rustup -h
rustup 1.5.0 (92d0d1e9e 2017-06-24)
The Rust toolchain installer

USAGE:
    rustup.exe [FLAGS] [SUBCOMMAND]

FLAGS:
    -v, --verbose    Enable verbose output
    -h, --help       Prints help information
    -V, --version    Prints version information

SUBCOMMANDS:
    show           Show the active and installed toolchains
    update         Update Rust toolchains and rustup
    default        Set the default toolchain
    toolchain      Modify or query the installed toolchains
    target         Modify a toolchain's supported targets
    component      Modify a toolchain's installed components
    override       Modify directory toolchain overrides
    run            Run a command with an environment configured
for a given toolchain
    which          Display which binary will be run for a given
command
    doc            Open the documentation for the current toolch
ain
    self           Modify the rustup installation
    set            Alter rustup settings
    completions    Generate completion scripts for your shell
    help           Prints this message or the help of the given
subcommand(s)
```

```
DISCUSSION:
    rustup installs The Rust Programming Language from the offic
ial
    release channels, enabling you to easily switch between stab
le,
    beta, and nightly compilers and keep them updated. It makes
    cross-compiling simpler with binary builds of the standard l
ibrary
    for common platforms.

    If you are new to Rust consider running `rustup doc --book`
to
    learn Rust.
```

根据提示, 使用 `rust help <command>` 来查看子命令的帮助。

`rustup doc --book` 会打开英文版的 The Rust Programming Language。

常用命令

`rustup default <toolchain>` 配置默认工具链。

`rustup show` 显示当前安装的工具链信息。

`rustup update` 检查安装更新。

`rustup toolchain [SUBCOMMAND]` 配置工具链

> - `rustup toolchain install <toolchain>` 安装工具链。
> - `rustup toolchain uninstall <toolchain>` 卸载工具链。
> - `rustup toolchain link <toolchain-name> "<toolchain-path>"` 设置自定义工具链。
>
> 其中标准的 `<toolchain>` 具有如下的形式
>
> ```
> `<channel>[-<date>][-<host>]`
> <channel>       = stable|beta|nightly|<version>
> <date>          = YYYY-MM-DD
> <host>          = <target-triple>
> ```
>
> 如 `stable-x86_64-pc-windows-msvc` `nightly-2017-7-25` `1.18.0` 等都是合法的toolchain名称。

`rustup override [SUBCOMMAND]` 配置一个目录以及其子目录的默认工具链

> 使用 `--path <path>` 指定目录或在某个目录下运行以下命令
>
> - `rustup override set <toolchain>` 设置该目录以及其子目录的默认工具链。
> - `rustup override unset` 取消目录以及其子目录的默认工具链。
>
> 使用 `rustup override list` 查看已设置的默认工具链。

`rustup target [SUBCOMMAND]` 配置工具链的可用目标

> - `rustup target add <target>` 安装目标。
> - `rustup target remove <target>` 卸载目标。
> - `rustup target add --toolchain <toolchain> <target>` 为特定工具链安装目标。

`rustup component` 配置 rustup 安装的组件

> - `rustup component add <component>` 安装组件
> - `rustup component remove <component>` 卸载组件
> - `rustup component list` 列出可用组件
>
> 常用组件：
>
> - Rust 源代码 `rustup component add rust-src`
> - Rust Langular Server (RLS) `rustup component add rls`

编辑器

本章描述几种常用编辑器针对 Rust 开发环境的配置。

前期准备

下载 Rust 源代码（供 racer 使用）

从github下载

`git clone https://github.com/rust-lang/rust.git`

从官网下载源代码包

下载地址： `https://static.rust-lang.org/dist/rustc-nightly-src.tar.gz`

使用rustup下载（推荐）

使用rustup获取源码最大的好处在于可以使用 `rustup update` 随时获取最新版源码，~~而且特别省事,~~执行以下命令获取源码

```
rustup component add rust-src
```

racer

racer是一个由rust的爱好者提供的rust自动补全和语法分析工具，被用来提供基本的补全功能和定义跳转功能。其本身完全由rust写成，补全功能已经比较完善了。

我们可以通过如下的方式获取它：

cargo自动安装

在rust 1.5版本以后，其安装包自带的cargo工具已经支持了cargo install命令，这个命令可以帮助我们通过简单的方式获取到 `racer` 的最新版。

你可以通过以下命令安装 `racer` 最新版，目前已知在Linux、Unix和Windows上适用

```
cargo install racer
```

编译安装

事实上我更推荐有条件的用户通过这种方式安装，因为自己实战操作一遍总是有些收获的。~~(帅气可爱的DCjanus表示怀疑)~~

下载源码

首先，我们需要下载racer的源码

```
git clone https://github.com/phildawes/racer.git
```

进行编译

然后，进入目录然后进行编译

```
cd racer && cargo build --release
```

这样，我们会得到racer的二进制文件在 `target/release/racer` 目录

设置环境变量

为了对Rust标准库进行补全，racer需要获取Rust源码路径。

设置名为 `RUST_SRC_PATH` 的环境变量为 `[path_to_your_rust_source]/src`

其中 `[path_to_your_rust_source]` 表示源码所在文件夹，使用rustup获取Rust源码的情况下 `[path_to_your_rust_source]` 默认为 `~/.multirust/toolchains/[your-toolchain]/lib/rustlib/src/rust/src`

测试

请重新打开终端，并进入到关闭之前的路径。 执行如下代码： linux:

```
./target/release/racer complete std::io::B
```

windows:

```
target\release\racer complete std::io::B
```

你将会看到racer的提示，这表示racer已经执行完成了。

安装 rustfmt

`cargo install rustfmt`

Rust Langular Server (RLS)

`Rust Langular Server` (下文简称 `RLS`)可以为很多IDE或编辑器提供包括不限于自动补全、跳转定义、重命名、跳转类型的功能支持。

使用rustup安装步骤如下:

1. 保证 `rustup` 为最新版

   ```
   rustup self update
   ```

2. 升级工具链(并不要求设置 `nightly` 为默认，但需要保证安装了 `nightly` 工具链)

   ```
   rustup update nightly
   ```

3. 正式安装RLS

   ```
   rustup component add rls --toolchain nightly
   rustup component add rust-analysis --toolchain nightly
   rustup component add rust-src --toolchain nightly
   ```

4. 设置环境变量 如果在安装Racer时没有设置名为 `RUST_SRC_PATH` 的环境变量，请参考前文进行设置。

截至当前(**2017**年**7**月**15**日)，`RLS`仍然处于**alpha**阶段，随着项目变动，安装步骤可能会由较大变化，本文中提及的**RLS**安装方法可能在较短的时间内过时，建议跟随官方安装指导进行安装。

该项目托管地址:**https://github.com/rust-lang-nursery/rls**

vim/GVim安装配置

本节介绍vim的Rust支持配置，在阅读本节之前，我们假定你已经拥有了一个可执行的rustc程序，并编译好了racer。

我的vim截图

应邀而加

使用vundle

vundle是vim的一个插件管理工具，基本上算是本类当中最为易用的了。 首先我们需要安装它

linux or OS X

```
mkdir -p  ~/.vim/bundle/
git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

windows

1. 首先找到你的gvim的安装路径，然后在路径下找到vimfiles文件夹
2. 在这个文件夹中将vundle库克隆到vimfiles/bundle/目录下的Vundle.vim文件夹中

启用rust支持

下载源码

首先，你需要下载rust-lang的源码文件，并将其解压到一个路径下。 这个源码文件我们可以从rust官网下载到，请下载你对应平台的文件。 然后将其解压到一个目录下，并找到其源码文件中的 `src` 目录。 比如我们解压源码包到 `C:\\rust-source\` ，那么我们需要的路径就是 `C:\\rust-source\src` ，记好这个路径，我们将在下一步用到它。

修改vim配置

首先找到你的vimrc配置文件，然后在其中添加如下配置

```
set nocompatible
filetype off
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()

Plugin 'VundleVim/Vundle.vim'
Plugin 'racer-rust/vim-racer'
Plugin 'rust-lang/rust.vim'

call vundle#end()

filetype on
```

然后为了让配置生效，我们重启我们的(g)vim，然后在vim里执行如下命令

```
:PluginInstall
```

这里vundle会自动的去仓库里拉取我们需要的文件，这里主要是vim-racer和rust.vim两个库。

更多的配置

为了让我们的vim能正常的使用，我们还需要在vimrc配置文件里加入一系列配置，

```
"" 开启rust的自动reformat的功能
let g:rustfmt_autosave = 1

"" 手动补全和定义跳转
set hidden
"" 这一行指的是你编译出来的racer所在的路径
let g:racer_cmd = "<path-to-racer>/target/release/racer"
"" 这里填写的就是我们在1.2.1中让你记住的目录
let $RUST_SRC_PATH="<path-to-rust-srcdir>/src/"
```

使用 YouCompleteMe

YouCompleteMe 是 vim 下的智能补全插件, 支持 C-family, Python, Rust 等的语法补全, 整合了多种插件, 功能强大. Linux 各发行版的官方源里基本都有软件包, 可直接安装. 如果有需要进行编译安装的话, 可参考官方教程

让 YCM 支持 Rust 需要在安装 YCM 过程中执行 ./install.py 时加上 --racer-completer, 并在 .vimrc 中添加如下设置

```
let g:ycm_rust_src_path="<path-to-rust-srcdir>/src/"
"" 一些方便的快捷键
""" 在 Normal 模式下, 敲 <leader>jd 跳转到定义或声明(支持跨文件)
nnoremap <leader>jd :YcmCompleter GoToDefinitionElseDeclaration<
CR>
""" 在 Insert 模式下, 敲 <leader>; 补全
inoremap <leader>; <C-x><C-o>
```

总结

经过不多的配置，我们得到了如下功能：

1. 基本的c-x c-o补全 (使用 YCM 后，能做到自动补全)
2. 语法着色
3. gd跳转到定义

总体来看支持度并不高。

lib.rs + (~/rust/iron/src) - GVIM

文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) 帮助(H)

```
//! web frameworks are instead addressed by the much simpler Modifier and Plugin
//! systems.
//!
//! Modifiers allow external code to manipulate Requests and Response in an ergonomic
//! fashion, allowing third-party extensions to get the same treatment as modifiers
//! defined in Iron itself. Plugins allow for lazily-evaluated, automatically cached
//! extensions to Requests and Responses, perfect for parsing, accessing, and
//! otherwise lazily manipulating an http connection.
//!
//! Middleware are only used when it is necessary to modify the control flow of a
//! Request flow, hijack the entire handling of a Request, check an incoming
//! Request, or to do final post-processing. This covers areas such as routing,
//! mounting, static asset serving, final template rendering, authentication, and
//! logging.
//!
//! Iron comes with only basic modifiers for setting the status, body, and various
//! headers, and the infrastructure for creating modifiers, plugins, and
//! middleware. No plugins or middleware are bundled with Iron.
//!

// Stdlib dependencies
#[macro_use] extern crate log;

// Third party packages
extern crate hyper;
extern crate typemap as tmap;
extern crate plugin;
extern crate error as err;
extern crate url;
extern crate num_cpus;
extern crate conduit_mime_types as mime_types;
#[macro_use]
extern crate lazy_static;

use std::net::IpAddr
                IpAddr
// Request +    Ipv4Addr
pub use reque   Ipv6Addr
pub use respo   Ipv6MulticastScope
                SocketAddr
// Middleware   SocketAddrV4
pub use middl   SocketAddrV6         ware, AfterMiddleware, AroundMiddleware,
                ToSocketAddrs        in};
                TcpStream
// Server       TcpListener
pub use iron:   Incoming
                UdpSocket
// Extensions   AddrParseError
pub use typem   Shutdown
                LookupHost
// Headers      lookup_host
pub use hyper   lookup_addr
pub use hyper::header::Headers;

// Expose `Pluggable` as `Plugin` so users can do `use iron::Plugin`.
pub use plugin::Pluggable as Plugin;

// Expose modifiers.
pub use modifier::Set;

// Errors
-- 全能补全 (^O^N^P) 匹配 1 / 17
```

额外的

Q1. 颜色好挫

A1. 我推荐一个配色，也是我自己用的 molokai

更详细内容可以参见我的vimrc配置，当然，我这个用的是比较老的版本的vundle，仅供参考。

Have a nice Rust !

Emacs

本节介绍 Emacs (Version 24) 的 Rust 配置，假设你已经安装好了 Emacs，并且有使用 Emacs 的经验。具体的安装和使用说明，见网上相关文档，在此不赘述。

另外，本节的例子是在 Mac OS 上，在 Linux 上面基本一样。

Windows的Emacs用户仅作参考。

简介

Emacs 的 rust-mode 提供了语法高亮显示和 elisp 函数，可以围绕 Rust 函数定义移动光标。有几个插件提供了附加的功能，如自动补全和动态语法检查。

main.rs

```
1  use std::thread;
2  use std::sync::Mutex;
3
4  fn main() {
5      let mut data = Mutex::new(vec![1u32, 2, 3]);
6
7      for i in 0..3 {
8          let data = data.lock().unwrap();
9          thread::spawn(move || {
10             data[i] += 1;
11         });
12     }
13
14     thread::sleep_ms(50);
15 }

U:---  main.rs        All L14     (Rust ElDoc Undo-Tree yas FlyC* company He
pub fn sleep_ms(ms: u32) {
```

安装插件

首先，需要将 melpa 代码库添加到你的插件列表中，才能安装 Rust 需要的插件。将下面的代码片段加入你的 `~/.emacs.d/init.el` 文件中。

```
;; Add melpa repository to archives
(add-to-list 'package-archives
    '("melpa" . "http://melpa.milkbox.net/packages/") t)

;; Initialize packages
(package-initialize)
```

运行下面的命令，更新插件列表。

- M-x eval-buffer
- M-x package-refresh-contents

然后，就可以安装插件，在 Emacs 中使用 Rust 了。运行 `M-x package-list-packages`，用 `i` 标记下述插件进行安装，当所有的插件选择好了之后，用 `x` 执行安装。

- company
- company-racer
- racer
- flycheck
- flycheck-rust
- rust-mode

将下面的代码片段加入你的 `~/.emacs.d/init.el` 文件：

```
;; Enable company globally for all mode
(global-company-mode)

;; Reduce the time after which the company auto completion popup
 opens
(setq company-idle-delay 0.2)

;; Reduce the number of characters before company kicks in
(setq company-minimum-prefix-length 1)
;; Set path to racer binary
(setq racer-cmd "/usr/local/bin/racer")
```

```
;; Set path to rust src directory
(setq racer-rust-src-path "/Users/YOURUSERNAME/.rust/src/")

;; Load rust-mode when you open `.rs` files
(add-to-list 'auto-mode-alist '("\\.rs\\'" . rust-mode))

;; Setting up configurations when you load rust-mode
(add-hook 'rust-mode-hook

     '(lambda ()
     ;; Enable racer
     (racer-activate)

     ;; Hook in racer with eldoc to provide documentation
     (racer-turn-on-eldoc)

     ;; Use flycheck-rust in rust-mode
     (add-hook 'flycheck-mode-hook #'flycheck-rust-setup)

     ;; Use company-racer in rust mode
     (set (make-local-variable 'company-backends) '(company-race
r))

     ;; Key binding to jump to method definition
     (local-set-key (kbd "M-.") #'racer-find-definition)

     ;; Key binding to auto complete and indent
     (local-set-key (kbd "TAB") #'racer-complete-or-indent)))
```

```
main.rs
1 ?fn main() {
2 ?     let mut x = vec!["Hello", "world"];
3
4 ?     let y = &x[0];
5  }

U:---  main.rs        All L2     (Rust ElDoc Undo-Tree yas FlyC:0/3 company Helm Projectil
variable does not need to be mutable, #[warn(unused_mut)] on by default
```

配置 Racer

Racer 需要 Rust 的源代码用于自动补全。

- git clone https://github.com/rust-lang/rust.git ~/.rust
- 重新启动 Emacs 并打开一个 Rust 源代码文件。

```
main.rs
1  use std::fs::
2              DirBuilderStruct
   fn main() { DirEntryStruct
       let mut FileStruct          orld"];
               FileTypeStruct
       let y = MetadataStruct
   }           OpenOptionsStruct
               PathExtTrait
               PermissionsStruct
               ReadDirStruct
               WalkDirStruct

U:**-  main.rs        Top L1     (Rust ElDoc Undo-Tree yas FlyC:2/0 company-racer Helm Pro
pub
```

结论

现在，可以在 Emacs 中编辑 Rust 源代码文件了。功能总结如下：

- 语法高亮显示和自动缩进
- 自动补全
- 动态语法错误检查
- 跳转到函数定义
- 内嵌文档

main.rs

```
1  use std::thread;
2
3 ?fn main() {
4      //
5      thread::spawn(|| {
6          println!("Hello from a thread!");
7      });
8  }
```

```
-:**-  main.rs        All L4     (Rust ElDoc Undo-Tree yas FlyC:0/1
```

注释

1. 本节的内容适用于 Emacs Version 24；版本 23 的配置方法不同；版本 22 及以下不支持。
2. MacOS 自带的 Emacs 版本是 22，版本 24 可以从这里下载。

VS Code 安装配置

VS Code 是微软出的一款开源代码编辑器，秉承了微软在IDE领域的一惯优秀基因，是一款潜力相当大的编辑器/IDE。

VScode 目前也对 Rust 也有良好的支持。

下载 VScode

请打开官网 https://code.visualstudio.com/ 下载编辑器。

依赖

如本章第一节所述，准备好 `racer`，`rust 源代码`，`rustfmt`，`rls` 这四样东西，并且配置好相应的环境变量，此不赘述。

安装 Rust 扩展 Rust

1. 打开 VScode 编辑器；
2. 按 Ctrl + p 打开命令面板；
3. 在编辑器中上部浮现出的输入框中，输入 `ext install vscode-rust`，会自动搜索可用的插件，搜索出来后，点击进行安装；
4. 使用 `VScode` 打开任意一个 `.rs` 文件，插件首次启动会自动引导用户完成配置。

注:推荐使用RLS模式，即使用Rust Langular Server提供各项功能支持

Atom

本文是rust的Atom编辑器配置。 横向对比一下，不得不说，Atom无论在易用性还是界面上都比前辈们要好的很多，对于Rust的配置，也是基本上可以做到开箱即用。 虽然本文独占一小节，但是其实能写的东西也就了了。

- 自行配置
- 使用tokamak

自行配置

准备工作

首先，你需要一个可执行的rustc编译器，一个cargo程序，一个已经编译好的racer程序和一份已经解压好的rust源码。 我们假定你已经将这三个程序安装完毕，并且能够自由的从命令行里调用他们。

另外，本文不讲解如何安装Atom，需要新安装的同学请自行前往项目主页安装。

ps:无论是windows用户还是*nix用户都需要将以上三个程序加入你的PATH(Windows下叫Path)环境变量里。

需要安装的插件包

打开Atom，按Ctrl+Shift+p，搜索preference，打开Atom的配置中心，选择install选项卡。

依次安装 `rust-api-docs-helper` / `racer` / `language-rust` / `linter-rust` / `linter` 。

这里要单独说的一个就是linter，这是一个基础的lint组件包，atom的很多以linter为前缀的包都会依赖这个包，但是Atom并不会为我们自动的安装，因此需要我们自己去安装。

一点配置

以上，我们安装好了几个组件包，但是不要着急去打开一个Rust文件。你可能还需要一点点的配置。这里，我们在配置中心里打开 `Packages` 选项卡，在 `Installed Packages` 里搜索racer，并点击其 `Setting` 。

这里需要将racer的可执行文件的绝对路径填入 `Path to the Racer executable` 里。同时，我们还需要将rust源码文件夹下的src目录加入到 `Path to the Rust source code directory` 里。

完成安装

好了，就是这么简单。你现在可以打开任意一个rust文件就会发现源码高亮已经默认打开了，编辑一下，racer也能自动补全，如果不能，尝试一下用 `F3` 键来显式地呼出racer的补全。

tokamak

tokamak 是一个使 atom 摇身一变为 rust IDE 的 atom 插件. 安装后 atom 即具有语法高亮, 代码补全与 Lint 等功能, 而且还有个不错的界面, 看起来确实像个 IDE. 你可以在 atom 中搜索 tokamak 并安装它.

Sublime

Sublime Text是一款非常有名的文本编辑器，其本身也具备强大的插件机制。通过配置各种插件可以在使用Sublime Text编辑rust代码时获得更加良好的支持。

本文主要展示在已经预装rust的Windows环境下的安装，如果您还没有安装rust，请先参照本书的安装章节安装rust。

安装

Sublime Text3安装

请在 Sublime Text3官网上选择适合当前机器版本的Sublime Text版本进行下载和安装。

rust的安装

请在rust官网的下载页面下载rust的源代码压缩包并在本地解压缩安装，在稍后的配置环节我们将会用到这个路径。如果国内下载速度过慢，可以考虑使用中科大的镜像下载rust源码包。

下载Rust并编译代码提示插件racer

具体安装和编译内容请查看本章第一节的安装准备，请牢记编译后的racer.exe文件路径，在稍后的配置环节中我们将用到它。

配置

Sublime Text3相关插件安装

安装Package Control

Sublime Text3在安装各种插件前需要先安装Package Control，如果您的编辑器已安装Package Control请跳过本段直接安装rust相关插件。

您可以查看Package Control官网学习如何安装。 也可以直接在编辑器中使用 `ctrl+~` 快捷键启动控制台，粘贴以下代码并回车进行安装。

```
import urllib.request,os,hashlib; h = '2915d1851351e5ee549c20394736b442' + '8bc59f460fa1548d1514676163dafc88'; pf = 'Package Control.sublime-package'; ipp = sublime.installed_packages_path(); urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandler()) ); by = urllib.request.urlopen( 'http://packagecontrol.io/' + pf.replace(' ', '%20')).read(); dh = hashlib.sha256(by).hexdigest(); print('Error validating download (got %s instead of %s), please try manual install' % (dh, h)) if dh != h else open(os.path.join( ipp, pf), 'wb' ).write(by)
```

rust相关插件

在编辑器下使用快捷键 `ctrl+shift+p` 启动命令行工具，输入Install Package按回车进入插件安装，选择或输入插件名称并回车即可完成插件的安装。

使用上述方式安装Rust插件(rust语法高亮)、RustAutoComplete(rust代码提示和自动补全插件)。

此时安装尚未完成，我们需要将本地的 racer.exe配置进RustAutoComplete插件中。打开编辑器顶端的Preferences选项卡，依次 Preferences->Package Settings->RustAutoComplete->Settings-User 来打开 RustAutoComplete 的配置文件，在文件中配置以下信息并保存。

```
{
  "racer": "E:/soft/racer-master/target/release/racer.exe",
  "search_paths": [    "E:/soft/rustc-1.7.0/src"    ]
}
```

其中racer是编译后的racer.exe程序的绝对路径。search_paths是rust源码文件下src目录的绝对路径。

编辑器重启后插件即可生效。

快速编译

Sublime本身支持多种编译系统，在Tools选项卡下的Build System中选择Rust或者Cargo作为编译系统，选中后使用快捷键 `ctrl+B` 即可对代码进行快速编译。

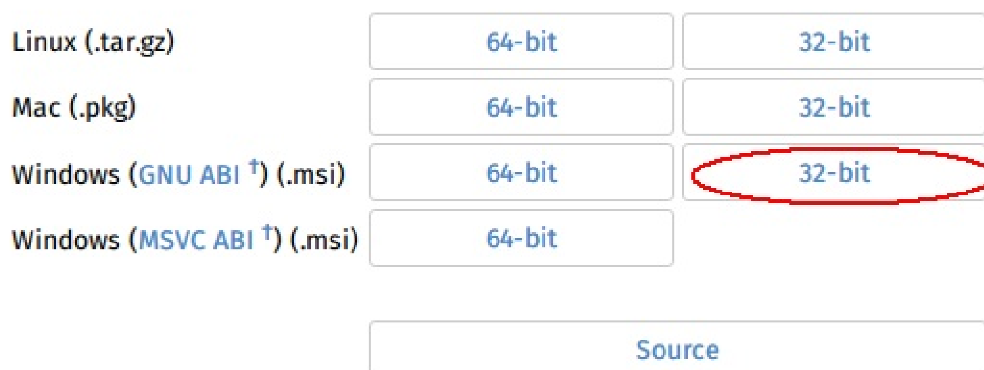
Visual Studio

本文是使用VisualRust和VS GDB Debugger / VisualGDB 完成在VisualStudio中，编辑和调试Rust程序。

安装Rust, Cargo

首先需要下载Rust, 下载地址https://www.rust-lang.org/downloads.html

这里一定要下windows GNU ABI的版本, 因为我们要用GDB来调试.



另外，机器上也需要安装Visual Studio2013或2015。 安装完Rust,打开命令行，执行 cargo install racer

```
Administrator: Command Prompt - cargo  install racer
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\marvin>cargo install racer
    Updating registry `https://github.com/rust-lang/crates.io-index`
```

Racer是用来做Rust自动完成的，会在VisualRust使用。这里我们使用rust编译的racer, 并不用VisualRust里自带的racer，因为它太旧了. 另外需要下载Rust源代码，设置 RUST_SRC_PATH为Rust源代码src的目录

```
PSModulePath=C:\Windows\system32\WindowsPowerShell\v1.0\Modu
PUBLIC=C:\Users\Public
RUST_SRC_PATH=V:\Script\Rust\src\src
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\Windows
```

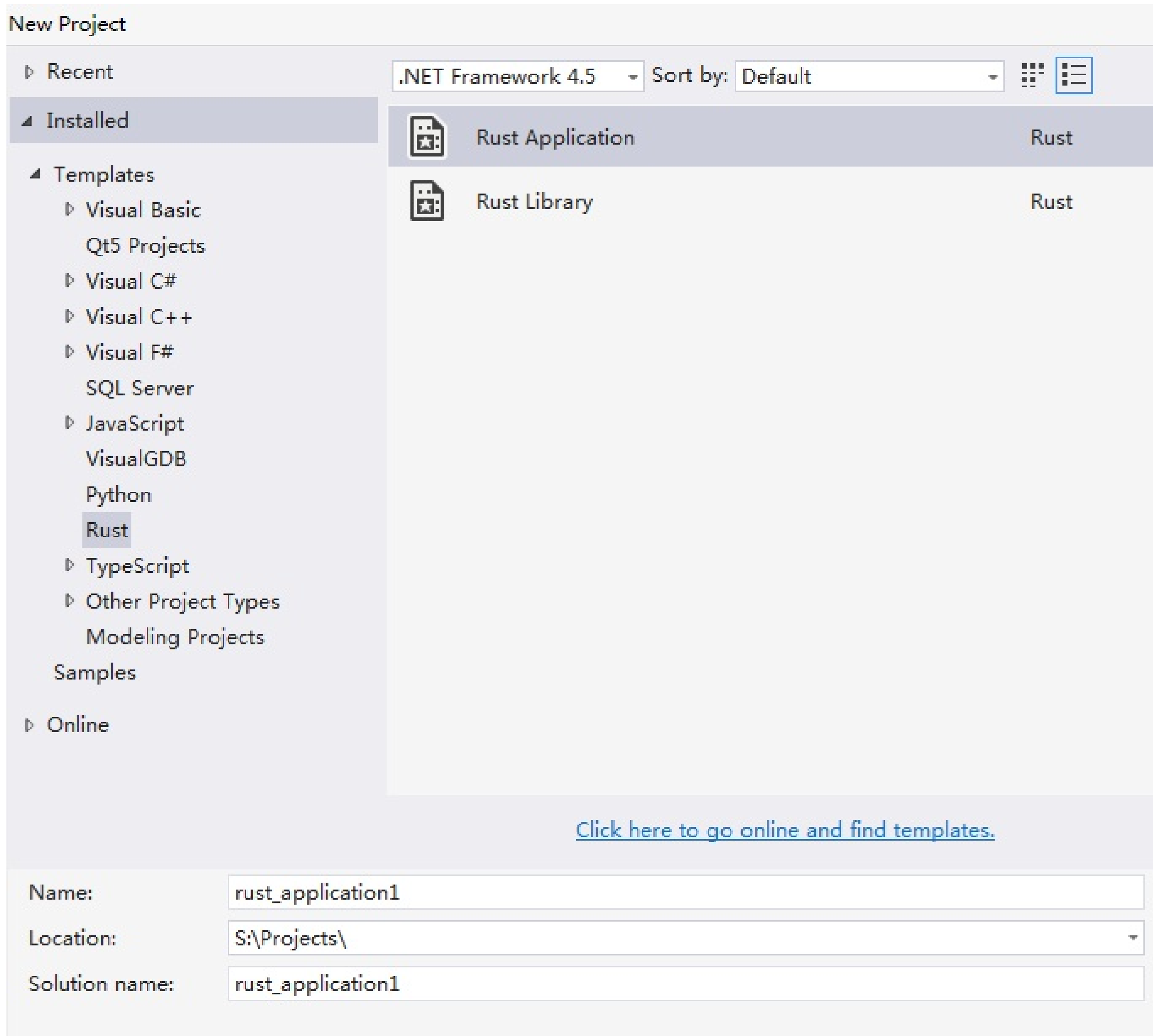
安装VisualRust和VS GDB Debugger

做完上述工作，就可以安装VisualRust和VS GDB Debugger,在这里下载
https://github.com/PistonDevelopers/VisualRust
https://visualstudiogallery.msdn.microsoft.com/35dbae07-8c1a-4f9d-94b7-bac16cad9c01

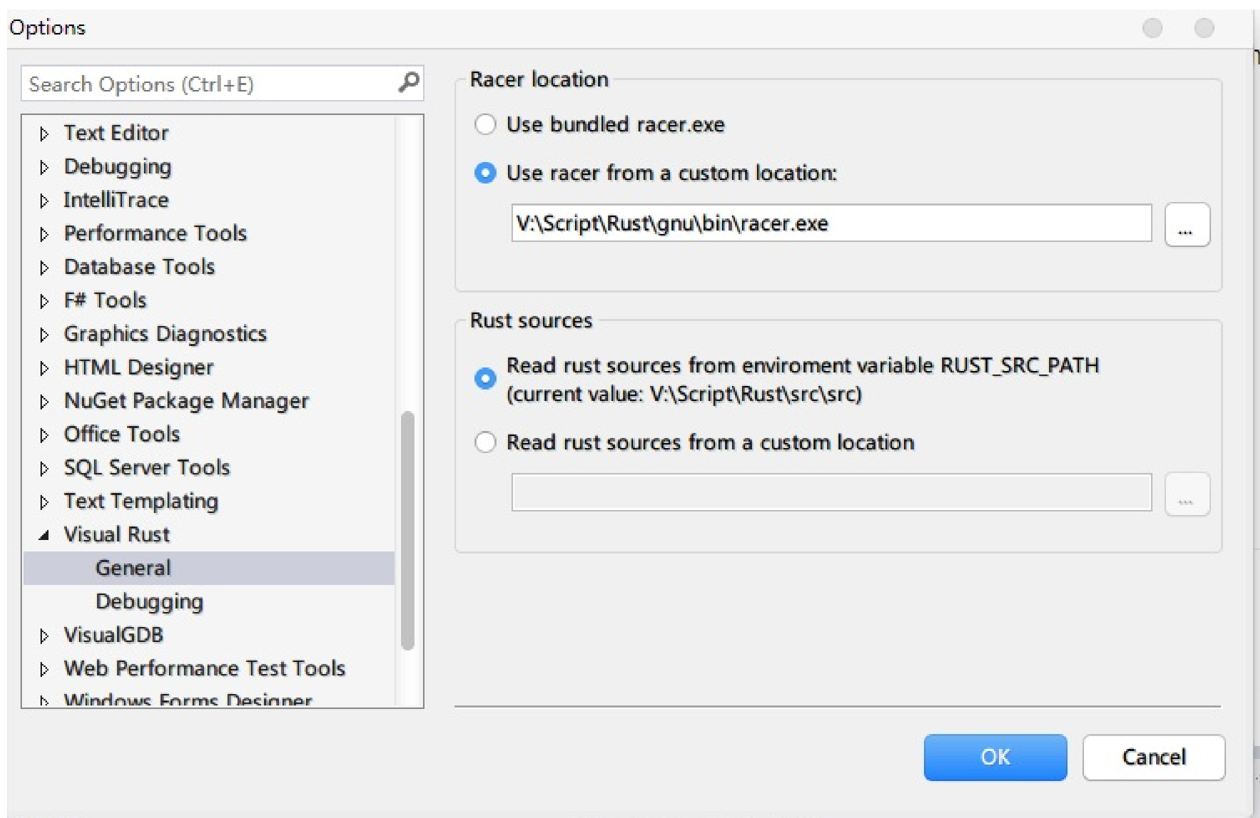
VisualGDB可在这里购买 http://www.visualgdb.com/

编译Rust项目

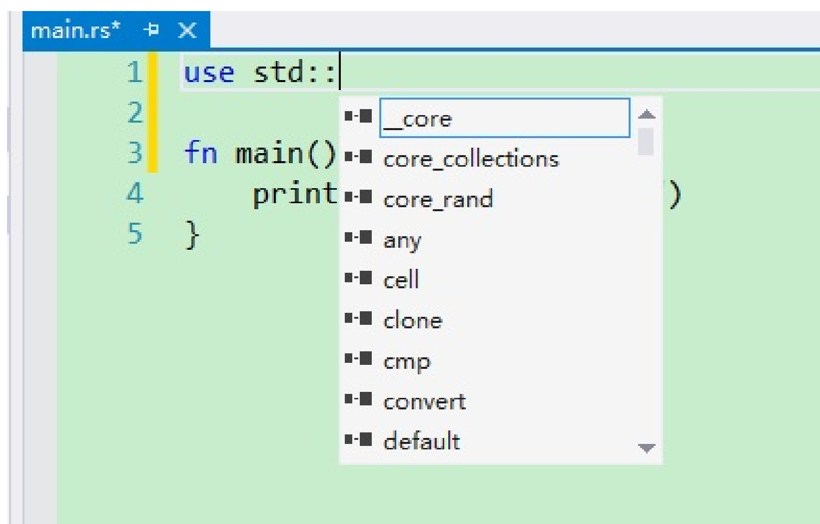
新建Rust项目



在tool, option里设置racer和rust_src_path



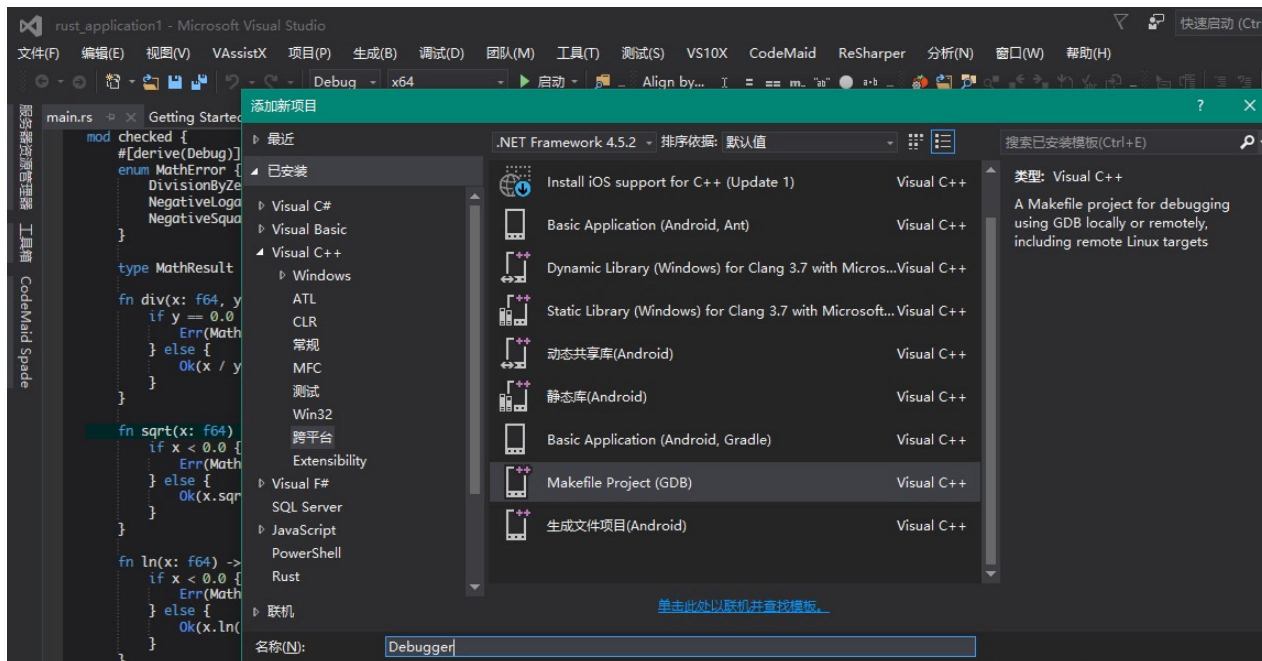
这时候就可以在写代码的时候就可以自动提示了。像下面这样



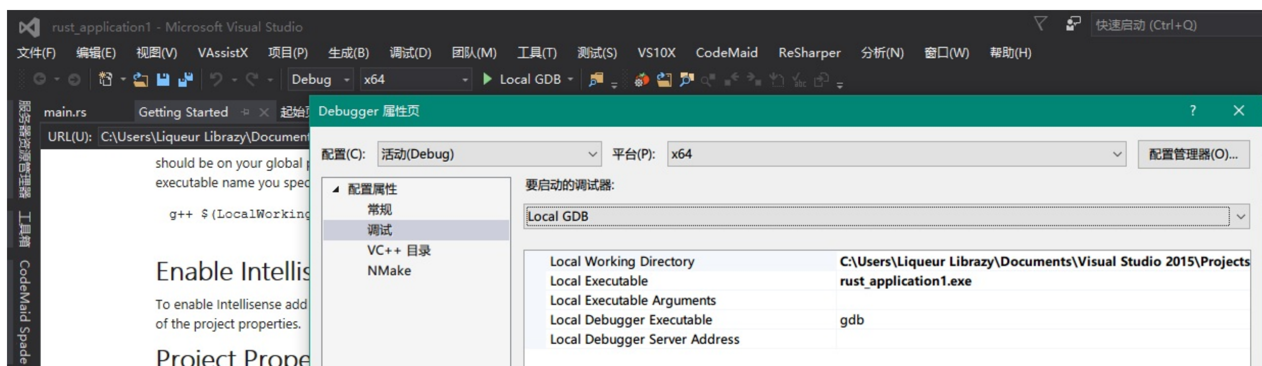
用VS GDB Debugger调试Rust项目

ok,愉快的开始你的Rust之旅吧。下面开始使用VS GDB Debugger调试Rust.

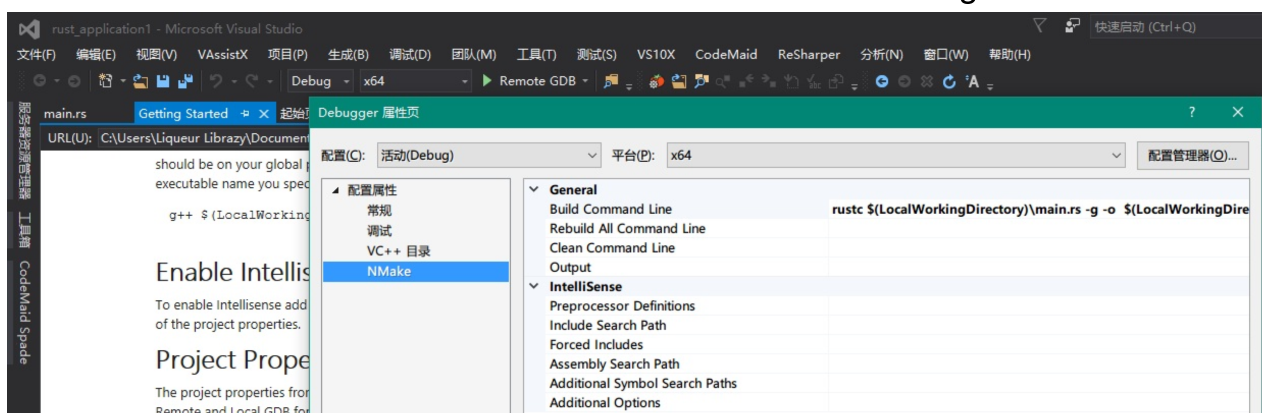
在解决方案中，添加GDB调试项目



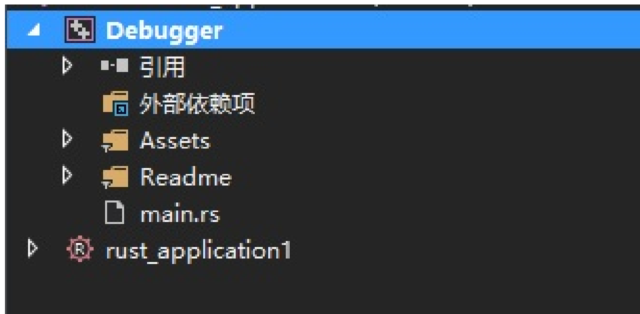
设置需要调试的程序所在的目录和文件名



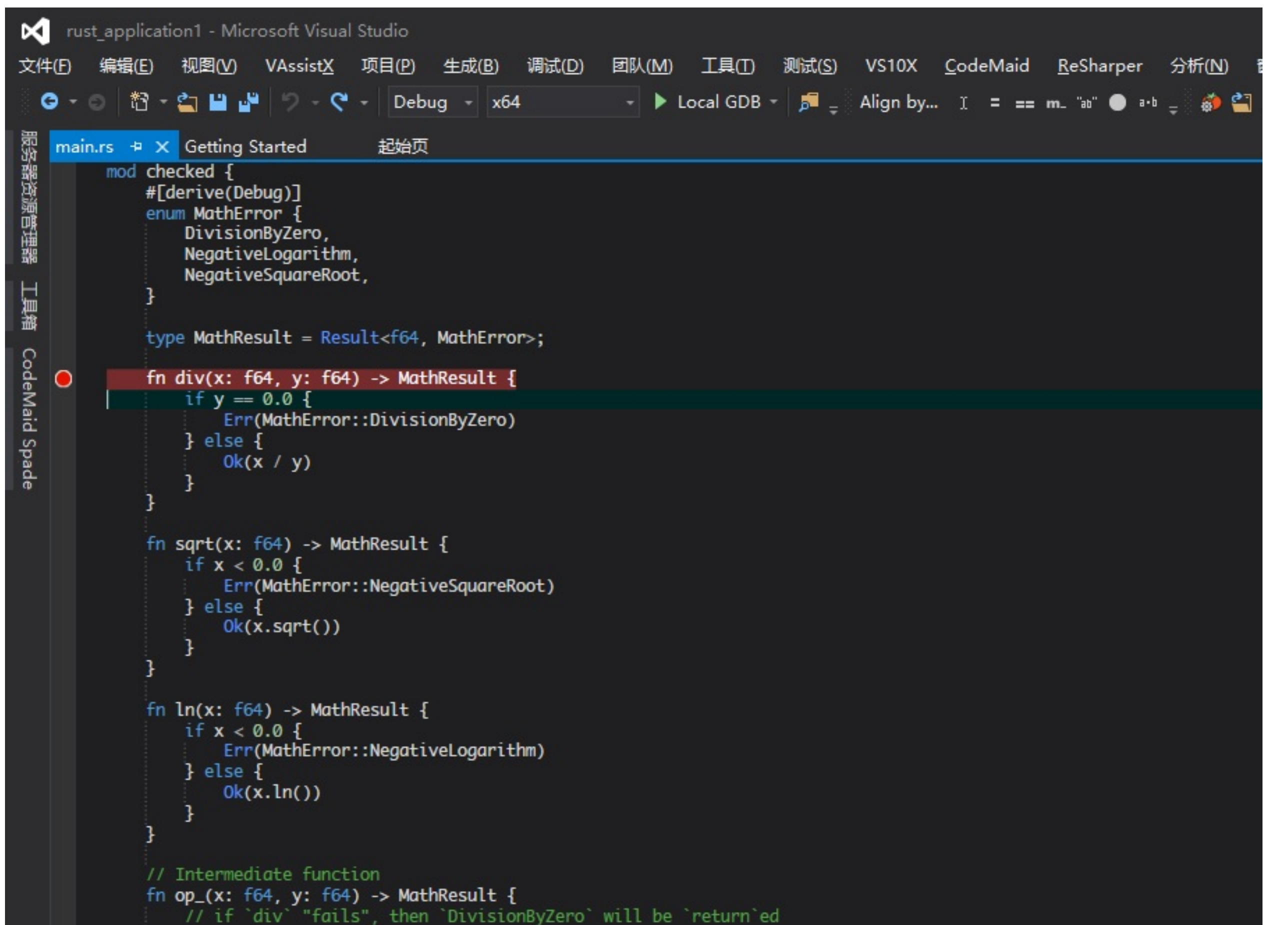
设置需要调试的程序的编译命令，此处用rustc，也可以使用cargo编译

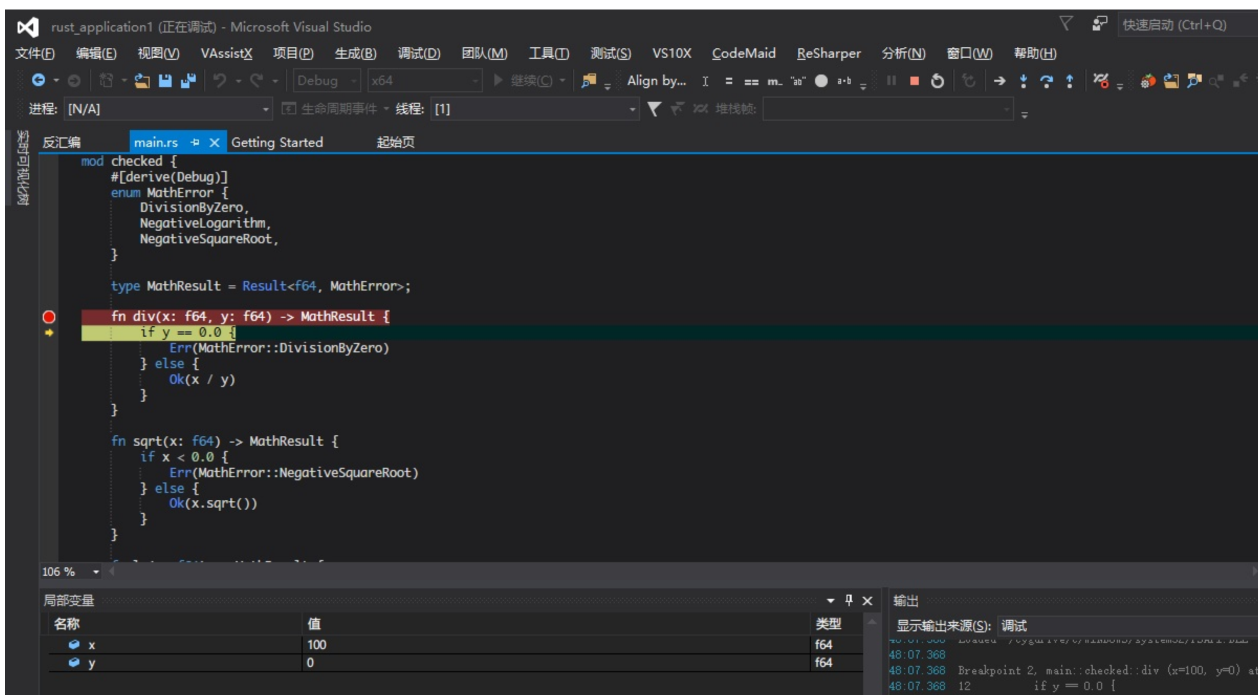


将需要调试的程序的源代码添加到项目目录下



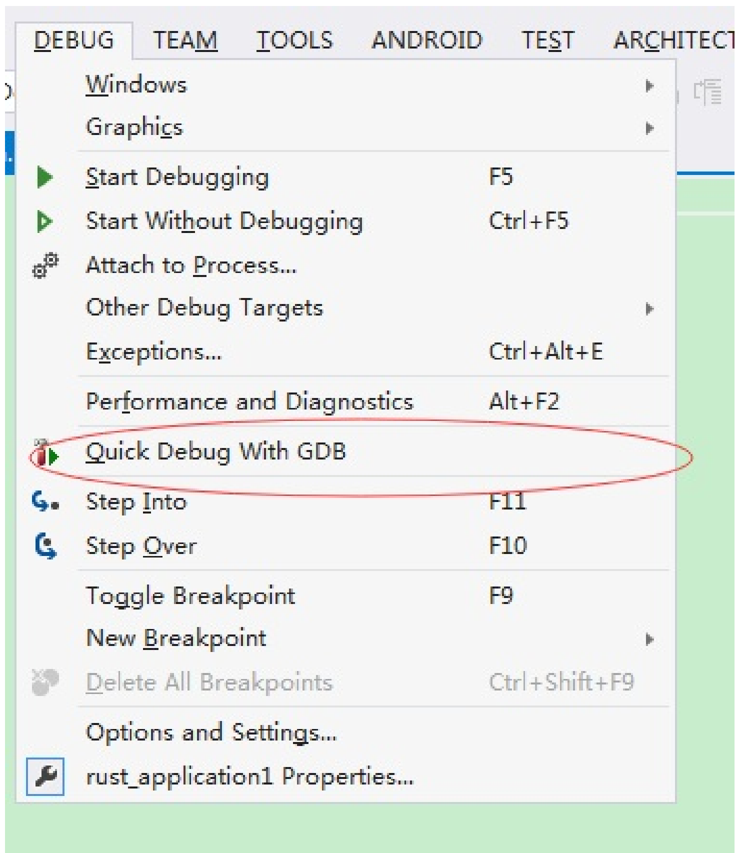
打开源代码文件并设置断点信息，将项目设置为启动项目并选择Local GDB即可开始调试



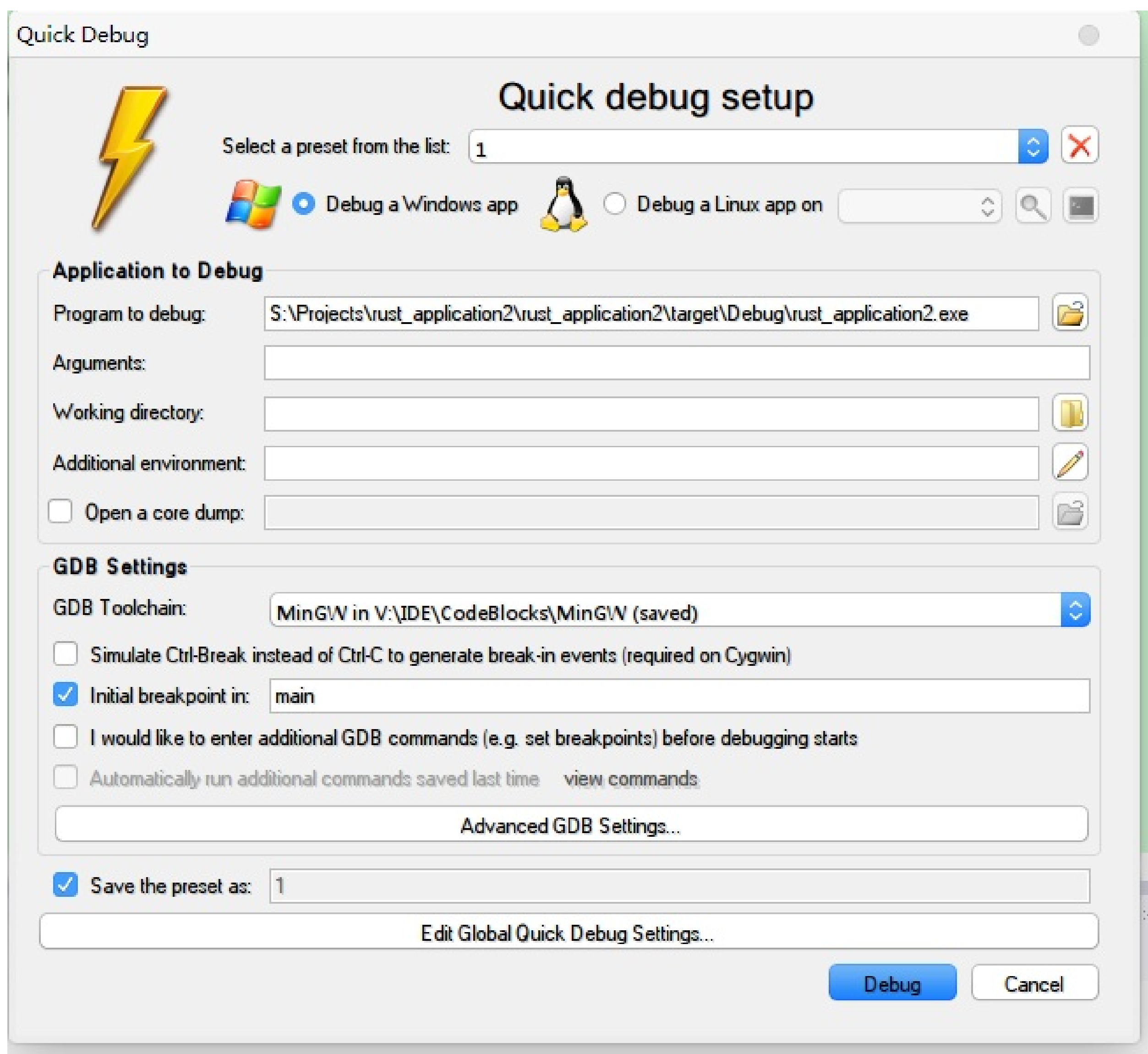


用VisualGDB调试Rust项目

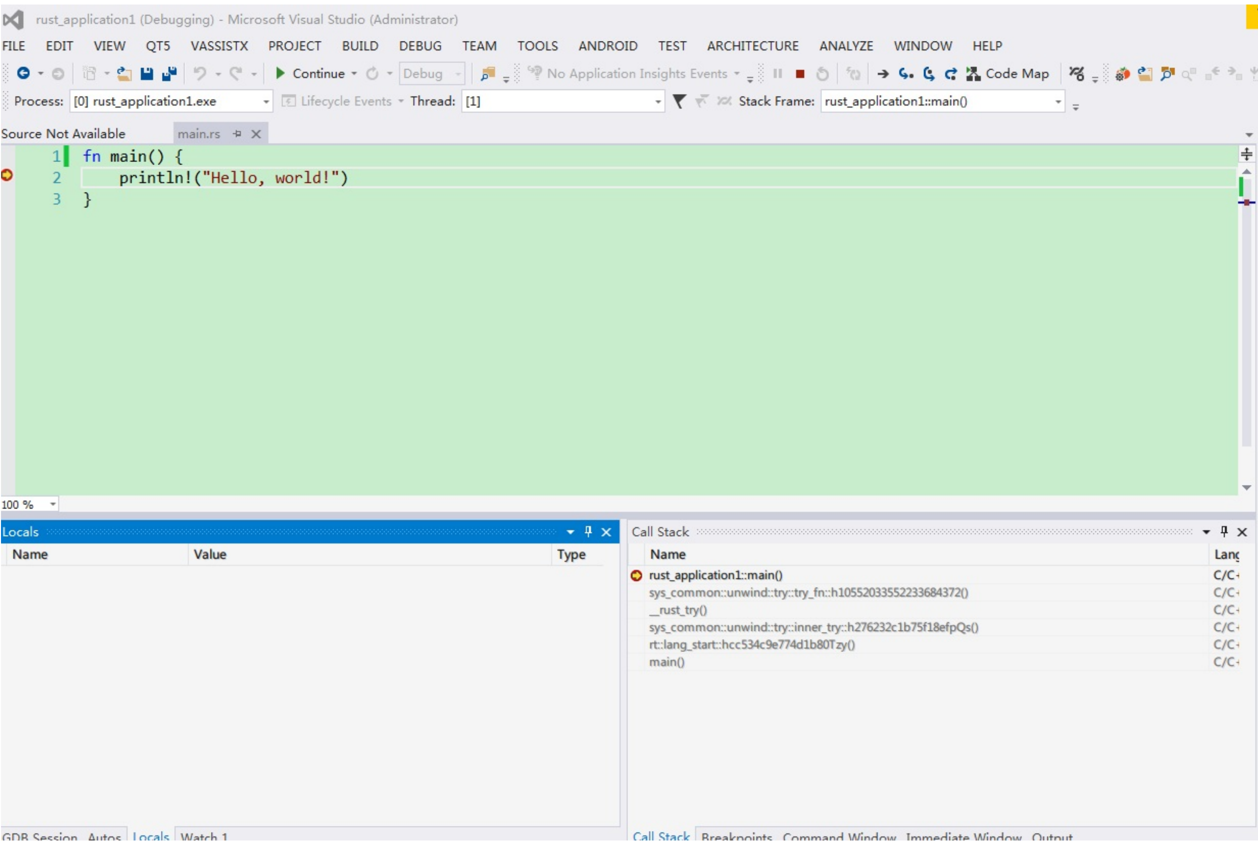
Build完Rust程序，点击debug, 选择quick debug with gdb



然后在里面选择MingW和exe的路径



点击Debug,开始你的调试生活吧


rust_application1 (Debugging) - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW QT5 VASSISTX PROJECT BUILD DEBUG TEAM TOOLS ANDROID TEST ARCHITECTURE ANALYZE WINDOW HELP
Continue
Debug
No Application Insights Events
Code Map
Process: [0] rust_application1.exe
Lifecycle Events
Thread: [1]
Stack Frame: rust_application1::main()
Source Not Available
main.rs
1 fn main() {
2 println!("Hello, world!")
3 }
100 %
Locals
Name
Value
Type
Call Stack
Name
Lang
rust_application1::main()
sys_common::unwind::try::try_fn::h10552033552233684372()
__rust_try()
sys_common::unwind::try::inner_try::h276232c1b75f18efpQs()
rt::lang_start::hcc534c9e774d1b80Tzy()
main()
C/C+
GDB Session Autos Locals Watch 1
Call Stack Breakpoints Command Window Immediate Window Output

Spacemacs

spacemacs，是一个给vimer的Emacs。

简介

spacemacs是一个专门给那些习惯vim的操作，同时又向往emacs的扩展能力的人。它非常适合我这种折腾过vim，配置过emacs的人，但同时也欢迎任何没有基础的新人使用。简单来说，它是一个开箱即用的Emacs！这对一个比很多人年龄都大的软件来说是一件极其不容易的事情。

安装

由于笔者自己在linux平台，并没有windows平台的经验，所以在这里便不献丑了，期待各位补充。另外，windows平台真的需要么，斜眼瞅向了Visual Studio。

Emacs安装

在*nix系统中，都不一定会默认安装了Emacs，就算安装了，也不一定是最新的版本。在这里，我强烈建议各位卸载掉系统自带的Emacs，因为你不知道系统给你安装的是个什么奇怪的存在，最遭心的，我碰见过只提供阉割版Emacs的linux发行版。

建议各位自己去emacs项目主页下载Emacs-24.5（本书写作时的最新版）极其以上版本，然后下载下来源码。至于Emacs的安装也非常简单，linux平台老三步。

```
./configure
make
sudo make install
```

什么？你没有make？没有GCC？缺少依赖？ 请安装它们……

Spacemacs安装

前面说了,Spacemacs就是个Emacs的配置文件库，因此我们可以通过非常简单的方式安装它：

```
git clone https://github.com/syl20bnr/spacemacs ~/.emacs.d
mv ~/.emacs ~/_emacs.backup
cd ~/.emacs.d
echo $(git describe --tags $(git rev-list --tags --max-count=1))
 | xargs git checkout
```

其中，后三行是笔者加的，这里必须要吐槽一下的是，Spacemacs的master分支实际上是极其落后而且有错误的！其目前的release都是从develop分支上打的tag。

因此，一！定！不！要！用！主！分！支！

最后，之所以要加最后一行，这是笔者安装的时候的release的一个小bug，没有这个文件的话,emacs并不会顺利的完成初始化。

好了，配置文件我们已经搞定了，接下来，启动你的emacs，spacemacs会自动的去网上下载你需要的插件安装包。另外，能自备梯子的最好，因为你要下的东西不大，但是这个网络确实比较捉急。

前期准备

为了让Spacemacs支持Rust，我们还需要一点小小的配置。首先，请参照前期准备，安装好你的racer。

在这里，强烈建议将racer的环境变量加入到系统变量中(通常他们在 `/etc/profile/` 里进行配置)并重新启动系统，因为真的有很多人直接点击emacs的图标启动它的，这样做很可能导致emacs并不继承自己的环境变量，这是很令人无奈的。

完成配置

修改标准的Spacemacs配置。

Spacemacs文档中提供了一份标准的spacemacs配置文件，你可以将其加入到你自己的 `~/.spacemacs` 文件中。

这里，我们需要修改的是其关于自定义插件的部分：

```
(defun dotspacemacs/layers ()
  "Configuration Layers declaration.
You should not put any user code in this function besides modify
ing the variable
values."
  (setq-default
   ;; Base distribution to use. This is a layer contained in the
 directory
   ;; `+distribution'. For now available distributions are `spac
emacs-base'
   ;; or `spacemacs'. (default 'spacemacs)
   dotspacemacs-distribution 'spacemacs
   ;; List of additional paths where to look for configuration l
ayers.
   ;; Paths must have a trailing slash (i.e. `~/.mycontribs/')
   dotspacemacs-configuration-layer-path '()
   ;; List of configuration layers to load. If it is the symbol
`all' instead
   ;; of a list then all discovered layers will be installed.
   dotspacemacs-configuration-layers
   '(
     ;; --------------------------------------------------------
--------
     ;; Example of useful layers you may want to use right away.
     ;; Uncomment some layer names and press <SPC f e R> (Vim st
yle) or
     ;; <M-m f e R> (Emacs style) to install them.
     ;; --------------------------------------------------------
--------
     auto-completion
     better-defaults
     git
     spell-checking
     syntax-checking
     version-control
     rust
     )
   ;; List of additional packages that will be installed without
```

```
 being
   ;; wrapped in a layer. If you need some configuration for the
se
   ;; packages then consider to create a layer, you can also put
 the
   ;; configuration in `dotspacemacs/config'.
   dotspacemacs-additional-packages '()
   ;; A list of packages and/or extensions that will not be inst
all and loaded.
   dotspacemacs-excluded-packages '()
   ;; If non-nil spacemacs will delete any orphan packages, i.e.
 packages that
   ;; are declared in a layer which is not a member of
   ;; the list `dotspacemacs-configuration-layers'. (default t)
   dotspacemacs-delete-orphan-packages t))

;; ...
;; 以下配置文件内容省略
;; ...
```

注意 `dotspacemacs-configuration-layers` 的配置和标准配置文件的不同。

将配置文件保存，然后重启你的emacs，当然，我们也可以按 `SPC f e R` 来完成重载配置文件的目的，然后你会发现emacs会开始下一轮下载，稍等其完成。

在上一步中，我们已经完成了对Racer的环境变量的配置，所以，现在你的Spacemacs已经配置完成了！这种简便的配置形式，几乎能和Atom抗衡了。

按键绑定

如下，spacemacs默认提供了几种按键绑定，但是，笔者并不觉得这些很好用，还是更喜欢用命令行。

Key Binding	Description
~SPC m c c~	compile project with Cargo
~SPC m c t~	run tests with Cargo
~SPC m c d~	generate documentation with Cargo
~SPC m c x~	execute the project with Cargo

尝试

现在开始，我们可以打开一个Cargo项目，并且去使用它了。你会惊讶的发现racer/flycheck/company这三个插件配合在一起的时候是那么的和谐简单。

快速上手

本章的目的在于快速上手(Quickstart)，对Rust语言建立初步的印象。 前面的章节中，我们已经安装好了Rust，配置好了编辑器，相信你一定已经跃跃欲试了。 注意: 本章的一些概念只需要大概了解就行，后续的章节将会有详细的讲解，但是本章的例子请务必亲自手敲并运行一遍。

下面，让我们开始动手写Rust程序吧!

ps：本章原始章节由 ee0703 书写的。因为内容不太满意，由 Naupio（N猫）重写了整个章节，并加入大量的内容。特别鸣谢 photino 提供的 rust-notes 。本章也有大量内容编辑自 Naupio（N猫） 创作中的 Rust 新书的快速入门章节。

Rust旅程

HelloWorld

按照编程语言的传统，学习第一门编程语言的第一个程序都是打印 Hello World！下面根据我们的步骤创建 Rust 的 Hello World！程序：

下面的命令操作，如果没有特别说明，都是在**shell**下运行。本文为了简单统一，所有例子都在 **win10** 的 **powershell** 下运行，所有命令都运行在 `ps:` 标识符之后

- 创建一个 Doing 目录和 helloworld.rs 文件

> ps: mkdir ~/Doing
> ps: cd ~/Doing
> ps: notepad helloworld.rs # 作者偏向于使用 sublime 作为编辑器
> ps: subl helloworld.rs # 本章以后使用 subl 代替 notepad

注意这里用的后缀名是.rs，一般编程语言的代码文件都有惯用的后缀名，比如：C语言是.c，java是.java，python是.py等等，请务必记住**Rust**语言的惯用后缀名是**.rs**（虽然用别的后缀名也能通过rustc的编译）。

- 在 helloworld.rs 文件中输入 Rust 代码

```rust
fn main() {
    println!("Hello World!");
}
```

- 编译 helloworld.rs 文件

> ps: rustc helloworld.rs
> ps: rustc helloworld.rs -O # 也可以选择优化编译

- 运行程序

> ps: ./helloworld.exe # windows 平台下需要加 .exe 后缀
> Hello World!

没有 `ps:` 前缀的表示为控制台打印输出。

我们已经用rust编写第一个可执行程序，打印出了'hello world!'，很酷，对吧！ 但是这段代码到底是什么意思呢，作为新手的你一定云里雾里吧，让我们先看一下这个程序：

1. 第一行中 fn 表示定义一个函数，main是这个函数的名字，花括号{}里的语句则表示这个函数的内容。
2. 名字叫做**main**的函数有特殊的用途，那就是作为程序的入口，也就是说程序每次都从这个函数开始运行。
3. 函数中只有一句 `println!("Hello World!");` ，这里 `println!` 是一个Rust语言自带的宏， 这个宏的功能就是打印文本(结尾会换行)，而"Hello World!"这个用引号包起来的东西是一个字符串，就是我们要打印的文本。
4. 你一定注意到了 `;` 吧， 在Rust语言中，分号 `;` 用来把语句分隔开，也就是说语句的末尾一般用分号做为结束标志。

HelloRust

- 创建项目 hellorust

> ps: cargo new hellorust --bin

- 查看目录结构

> ps: tree # win10 powershell 自带有 tree 查看文件目录结构的功能
> └─hellorust
> ---- └─src

这里显示的目录结构，在hellorust目录下有 src 文件夹和 Cargo.toml 文件，同时这个目录会初始化为 git 项目

- 查看Cargo.toml文件

> ps: cat Cargo.toml
> [package]
> name = "hellorust"
> version = "0.1."
> authors = ["YourName "]
> [dependencies]

- 编辑src目录下的main.rs文件

> ps: subl ./src/main.rs

cargo 创建的项目，在src目录下会有一个初始化的main.rs文件，内容为：

```
fn main() {
    println!("Hello, world!");
}
```

现在我们编辑这个文件，改为：

```
fn main() {
    let rust = "Rust";
    println!("Hello, {}!", rust);
}
```

这里的 `let rust = "Rust"` 是把 rust 变量绑定为 "Rust" ， `println!("Hello, {}!", rust);` 里把 rust 变量的值代入到 `"Hello, {}!"` 中的 `{}` 。

- 编译和运行

> ps: cargo build
> ps: cargo build --release # 这个属于优化编译
> ps: ./target/debug/hellorust.exe
> ps: ./target/release/hellorust.exe # 如果前面是优化编译，则这样运行
> ps: cargo run # 编译和运行合在一起
> ps: cargo run --release # 同上，区别是是优化编译的

变量绑定与原生类型

变量绑定

Rust 通过 let 关键字进行变量绑定。

```
fn main() {
    let a1 = 5;
    let a2:i32 = 5;
    assert_eq!(a1, a2);
    //let 绑定 整数变量默认类型推断是 i32

    let b1:u32 = 5;
    //assert_eq!(a1, b1);
    //去掉上面的注释会报错，因为类型不匹配
    //errer: mismatched types
}
```

这里的 assert_eq! 宏的作用是判断两个参数是不是相等的，但如果是两个不匹配的类型，就算字面值相等也会报错。

可变绑定

rust 在声明变量时，在变量前面加入 mut 关键字，变量就会成为可变绑定的变量。

```
fn main() {
    let mut a: f64 = 1.0;
    let b = 2.0f32;

    //改变 a 的绑定
    a = 2.0;
    println!("{:?}", a);

    //重新绑定为不可变
    let a = a;

    //不能赋值
    //a = 3.0;

    //类型不匹配
    //assert_eq!(a, b);
}
```

这里的 b 变量，绑定了 2.0f32。这是 Rust 里面值类型显式标记的语法，规定为 `value` + `type` 的形式。

例如：固定大小类型：

> 1u8 1i8
> 1u16 1i16
> 1u32 1i32
> 1u64 1i64

可变大小类型：

> 1usize 1isize

浮点类型：

> 1f32 1f64

let解构

为什么在 Rust 里面声明一个变量的时候要采用 let 绑定表达式？ 那是因为 let 绑定表达式的表达能力更强，而且 let 表达式实际上是一种模式匹配。

例如：

```
fn main() {
    let (a, mut b): (bool,bool) = (true, false);
    println!("a = {:?}, b = {:?}", a, b);
    //a 不可变绑定
    //a = false;

    //b 可变绑定
    b = true;
    assert_eq!(a, b);
}
```

这里使用了 bool，只有true和false两个值，通常用来做逻辑判断的类型。

原生类型

Rust内置的原生类型 (primitive types) 有以下几类：

- 布尔类型：有两个值 `true` 和 `false` 。
- 字符类型：表示单个Unicode字符，存储为4个字节。
- 数值类型：分为有符号整数 (`i8` , `i16` , `i32` , `i64` , `isize`)、 无符号整数 (`u8` , `u16` , `u32` , `u64` , `usize`) 以及浮点数 (`f32` , `f64`)。
- 字符串类型：最底层的是不定长类型 `str` ，更常用的是字符串切片 `&str` 和堆分配字符串 `String` ， 其中字符串切片是静态分配的，有固定的大小，并且不可变，而堆分配字符串是可变的。
- 数组：具有固定大小，并且元素都是同种类型，可表示为 `[T; N]` 。
- 切片：引用一个数组的部分数据并且不需要拷贝，可表示为 `&[T]` 。
- 元组：具有固定大小的有序列表，每个元素都有自己的类型，通过解构或者索引来获得每个元素的值。
- 指针：最底层的是裸指针 `*const T` 和 `*mut T` ，但解引用它们是不安全的，必须放到 `unsafe` 块里。
- 函数：具有函数类型的变量实质上是一个函数指针。
- 元类型：即 `()` ，其唯一的值也是 `()` 。

```
// boolean type
let t = true;
let f: bool = false;

// char type
let c = 'c';

// numeric types
let x = 42;
let y: u32 = 123_456;
let z: f64 = 1.23e+2;
let zero = z.abs_sub(123.4);
let bin = 0b1111_0000;
let oct = 0o7320_1546;
let hex = 0xf23a_b049;

// string types
let str = "Hello, world!";
let mut string = str.to_string();

// arrays and slices
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4];
let mut ten_zeros: [i64; 10] = [0; 10];

// tuples
let tuple: (i32, &str) = (50, "hello");
let (fifty, _) = tuple;
let hello = tuple.1;

// raw pointers
let x = 5;
let raw = &x as *const i32;
let points_at = unsafe { *raw };

// functions
fn foo(x: i32) -> i32 { x }
let bar: fn(i32) -> i32 = foo;
```

有几点是需要特别注意的：

- 数值类型可以使用 `_` 分隔符来增加可读性。
- Rust还支持单字节字符 `b'H'` 以及单字节字符串 `b"Hello"` ，仅限制于ASCII字符。此外，还可以使用 `r#"..."#` 标记来表示原始字符串，不需要对特殊字符进行转义。
- 使用 `&` 符号将 `String` 类型转换成 `&str` 类型很廉价，但是使用 `to_string()` 方法将 `&str` 转换到 `String` 类型涉及到分配内存，除非很有必要否则不要这么做。
- 数组的长度是不可变的，动态的数组称为Vec (vector)，可以使用宏 `vec!` 创建。
- 元组可以使用 `==` 和 `!=` 运算符来判断是否相同。
- 不多于32个元素的数组和不多于12个元素的元组在值传递时是自动复制的。
- Rust不提供原生类型之间的隐式转换，只能使用 `as` 关键字显式转换。
- 可以使用 `type` 关键字定义某个类型的别名，并且应该采用驼峰命名法。

```
// explicit conversion
let decimal = 65.4321_f32;
let integer = decimal as u8;
let character = integer as char;

// type aliases
type NanoSecond = u64;
type Point = (u8, u8);
```

数组、动态数组和字符串

数组和动态数组

数组 array

Rust 使用数组存储相同类型的数据集。 `[T; N]` 表示一个拥有 T 类型，N 个元素的数组。数组的大小是固定。

例子：

```
fn main() {
    let mut array: [i32; 3] = [0; 3];

    array[1] = 1;
    array[2] = 2;

    assert_eq!([1, 2], &array[1..]);

    // This loop prints: 0 1 2
    for x in &array {
        println!("{} ", x);
    }
}
```

动态数组 Vec

动态数组是一种基于堆内存申请的连续动态数据类型，拥有 O(1) 时间复杂度的索引、压入（push）、弹出（pop)。

例子：

```
//创建空Vec
let v: Vec<i32> = Vec::new();
//使用宏创建空Vec
let v: Vec<i32> = vec![];
//创建包含5个元素的Vec
let v = vec![1, 2, 3, 4, 5];
//创建十个零
let v = vec![0; 10];
//创建可变的Vec，并压入元素3
let mut v = vec![1, 2];
v.push(3);
//创建拥有两个元素的Vec，并弹出一个元素
let mut v = vec![1, 2];
let two = v.pop();
//创建包含三个元素的可变Vec，并索引一个值和修改一个值
let mut v = vec![1, 2, 3];
let three = v[2];
v[1] = v[1] + 5;
```

字符串

Rust 里面有两种字符串类型。`String` 和 `str` 。

&str

`str` 类型基本上不怎么使用，通常使用 `&str` 类型，它其实是 `[u8]` 类型的切片形式 `&[u8]` 。这是一种固定大小的字符串类型。 常见的的字符串字面值就是 `&'static str` 类型。这是一种带有 `'static` 生命周期的 &str 类型。

例子：

```
// 字符串字面值
let hello = "Hello, world!";

// 附带显式类型标识
let hello: &'static str = "Hello, world!";
```

String

`String` 是一个带有的 `vec:Vec<u8>` 成员的结构体，你可以理解为 `str` 类型的动态形式。 它们的关系相当于 `[T]` 和 `Vec<T>` 的关系。 显然 `String` 类型也有压入和弹出。

例子：

```
// 创建一个空的字符串
let mut s = String::new();
// 从 `&str` 类型转化成 `String` 类型
let mut hello = String::from("Hello, ");
// 压入字符和压入字符串切片
hello.push('w');
hello.push_str("orld!");

// 弹出字符。
let mut s = String::from("foo");
assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('f'));
assert_eq!(s.pop(), None);
```

结构体与枚举

结构体

结构体 (struct) 是一种记录类型，所包含的每个域 (field) 都有一个名称。 每个结构体也都有一个名称，通常以大写字母开头，使用驼峰命名法。 元组结构体 (tuple struct) 是由元组和结构体混合构成，元组结构体有名称， 但是它的域没有。当元组结构体只有一个域时，称为新类型 (newtype)。 没有任何域的结构体，称为类单元结构体 (unit-like struct)。 结构体中的值默认是不可变的，需要给结构体加上 `mut` 使其可变。

```
// structs
struct Point {
  x: i32,
  y: i32,
}
let point = Point { x: 0, y: 0 };

// tuple structs
struct Color(u8, u8, u8);
let android_green = Color(0xa4, 0xc6, 0x39);
let Color(red, green, blue) = android_green;

// A tuple struct's constructors can be used as functions.
struct Digit(i32);
let v = vec![0, 1, 2];
let d: Vec<Digit> = v.into_iter().map(Digit).collect();

// newtype: a tuple struct with only one element
struct Inches(i32);
let length = Inches(10);
let Inches(integer_length) = length;

// unit-like structs
struct EmptyStruct;
let empty = EmptyStruct;
```

一个包含 `..` 的 `struct` 可以用来从其它结构体拷贝一些值或者在解构时忽略一些域：

```
#[derive(Default)]
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point3d::default();
let point = Point3d { y: 1, ..origin };
let Point3d { x: x0, y: y0, .. } = point;
```

需要注意，Rust在语言级别不支持域可变性 (field mutability)，所以不能这么写：

```
struct Point {
    mut x: i32,
    y: i32,
}
```

这是因为可变性是绑定的一个属性，而不是结构体自身的。可以使用 `Cell<T>` 来模拟：

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);
```

此外，结构体的域对其所在模块 (mod) 之外默认是私有的，可以使用 `pub` 关键字将其设置成公开。

```
mod graph {
    #[derive(Default)]
    pub struct Point {
        pub x: i32,
        y: i32,
    }

    pub fn inside_fn() {
        let p = Point {x:1, y:2};
        println!("{}, {}", p.x, p.y);
    }
}

fn outside_fn() {
    let p = graph::Point::default();
    println!("{}", p.x);
    // println!("{}", p.y);
    // field `y` of struct `graph::Point` is private
}
```

枚举

Rust有一个集合类型，称为枚举 (enum)，代表一系列子数据类型的集合。 其中子数据结构可以为空-如果全部子数据结构都是空的，就等价于C语言中的enum。 我们需要使用 `::` 来获得每个元素的名称。

```
// enums
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

let x: Message = Message::Move { x: 3, y: 4 };
```

与结构体一样，枚举中的元素默认不能使用关系运算符进行比较 (如 `==`, `!=`, `>=`)， 也不支持像 `+` 和 `*` 这样的双目运算符，需要自己实现，或者使用 `match` 进行匹配。

枚举默认也是私有的，如果使用 `pub` 使其变为公有，则它的元素也都是默认公有的。 这一点是与结构体不同的：即使结构体是公有的，它的域仍然是默认私有的。这里的共有/私有仍然 是针对其定义所在的模块之外。此外，枚举和结构体也可以是递归的 (recursive)。

控制流(control flow)

If

If是分支 (branch) 的一种特殊形式，也可以使用 `else` 和 `else if` 。 与C语言不同的是，逻辑条件不需要用小括号括起来，但是条件后面必须跟一个代码块。 Rust中的 `if` 是一个表达式 (expression)，可以赋给一个变量：

```
let x = 5;

let y = if x == 5 { 10 } else { 15 };
```

Rust是基于表达式的编程语言，有且仅有两种语句 (statement)：

1. 声明语句 (declaration statement)，比如进行变量绑定的 `let` 语句。
2. 表达式语句 (expression statement)，它通过在末尾加上分号 `;` 来将表达式变成语句， 丢弃该表达式的值，一律返回unit `()` 。

表达式如果返回，总是返回一个值，但是语句不返回值或者返回 `()` ，所以以下代码会报错：

```
let y = (let x = 5);

let z: i32 = if x == 5 { 10; } else { 15; };
```

值得注意的是，在Rust中赋值 (如 `x = 5`) 也是一个表达式，返回unit的值 `()` 。

For

Rust中的 `for` 循环与C语言的风格非常不同，抽象结构如下：

```
for var in expression {
    code
}
```

其中 `expression` 是一个迭代器 (iterator)，具体的例子为 `0..10` (不包含最后一个值)， 或者 `[0, 1, 2].iter()` 。

While

Rust中的 `while` 循环与C语言中的类似。对于无限循环，Rust有一个专用的关键字 `loop` 。 如果需要提前退出循环，可以使用关键字 `break` 或者 `continue` ，还允许在循环的开头设定标签 (同样适用于 `for` 循环)：

```
'outer: loop {
   println!("Entered the outer loop");

   'inner: loop {
       println!("Entered the inner loop");
       break 'outer;
   }

   println!("This point will never be reached");
}

println!("Exited the outer loop");
```

Match

Rust中的 `match` 表达式非常强大，首先看一个例子：

```
let day = 5;

match day {
  0 | 6 => println!("weekend"),
  1 ... 5 => println!("weekday"),
  _ => println!("invalid"),
}
```

其中 `|` 用于匹配多个值， `...` 匹配一个范围 (包含最后一个值)，并且 `_` 在这里是必须的， 因为 `match` 强制进行穷尽性检查 (exhaustiveness checking)，必须覆盖所有的可能值。 如果需要得到 `|` 或者 `...` 匹配到的值，可以使用 `@` 绑定变量：

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

使用 `ref` 关键字来得到一个引用：

```
let x = 5;
let mut y = 5;

match x {
    // the `r` inside the match has the type `&i32`
    ref r => println!("Got a reference to {}", r),
}

match y {
    // the `mr` inside the match has the type `&i32` and is muta
ble
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

再看一个使用 `match` 表达式来解构元组的例子：

```
let pair = (0, -2);

match pair {
    (0, y) => println!("x is `0` and `y` is `{:?}`", y),
    (x, 0) => println!("`x` is `{:?}` and y is `0`", x),
    _ => println!("It doesn't matter what they are"),
}
```

`match` 的这种解构同样适用于结构体或者枚举。如果有必要，还可以使用 `..` 来忽略域或者数据：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}

enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    // 这里是 match 的 if guard 表达式，我们将在以后的章节进行详细介绍
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigge
r than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

此外，Rust还引入了 `if let` 和 `while let` 进行模式匹配：

```
let number = Some(7);
let mut optional = Some(0);

// If `let` destructures `number` into `Some(i)`, evaluate the block.
if let Some(i) = number {
    println!("Matched {:?}!", i);
} else {
    println!("Didn't match a number!");
}

// While `let` destructures `optional` into `Some(i)`, evaluate the block.
while let Some(i) = optional {
    if i > 9 {
        println!("Greater than 9, quit!");
        optional = None;
    } else {
        println!("`i` is `{:?}`. Try again.", i);
        optional = Some(i + 1);
    }
}
```

函数与方法

函数

要声明一个函数，需要使用关键字 `fn` ，后面跟上函数名，比如

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

其中函数参数的类型不能省略，可以有多个参数，但是最多只能返回一个值， 提前返回使用 `return` 关键字。Rust编译器会对未使用的函数提出警告， 可以使用属性 `#[allow(dead_code)]` 禁用无效代码检查。

Rust有一个特殊特性适用于发散函数 (diverging function)，它不返回：

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

其中 `panic!` 是一个宏，使当前执行线程崩溃并打印给定信息。返回类型 `!` 可用作任何类型：

```
let x: i32 = diverges();
let y: String = diverges();
```

匿名函数

Rust使用闭包 (closure) 来创建匿名函数：

```
let num = 5;
let plus_num = |x: i32| x + num;
```

其中闭包 `plus_num` 借用了它作用域中的 `let` 绑定 `num` 。如果要让闭包获得所有权，可以使用 `move` 关键字：

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;   // 闭包通过move获取了num的所有权

    add_num(5);
}

// 下面的num在被move之后还能继续使用是因为其实现了Copy特性
// 具体可见所有权(Owership)章节
assert_eq!(5, num);
```

高阶函数

Rust 还支持高阶函数 (high order function)，允许把闭包作为参数来生成新的函数：

```
fn add_one(x: i32) -> i32 { x + 1 }

fn apply<F>(f: F, y: i32) -> i32
    where F: Fn(i32) -> i32
{
    f(y) * y
}

fn factory(x: i32) -> Box<Fn(i32) -> i32> {
    Box::new(move |y| x + y)
}

fn main() {
    let transform: fn(i32) -> i32 = add_one;
    let f0 = add_one(2i32) * 2;
    let f1 = apply(add_one, 2);
    let f2 = apply(transform, 2);
    println!("{}, {}, {}", f0, f1, f2);

    let closure = |x: i32| x + 1;
    let c0 = closure(2i32) * 2;
    let c1 = apply(closure, 2);
    let c2 = apply(|x| x + 1, 2);
    println!("{}, {}, {}", c0, c1, c2);

    let box_fn = factory(1i32);
    let b0 = box_fn(2i32) * 2;
    let b1 = (*box_fn)(2i32) * 2;
    let b2 = (&box_fn)(2i32) * 2;
    println!("{}, {}, {}", b0, b1, b2);

    let add_num = &(*box_fn);
    let translate: &Fn(i32) -> i32 = add_num;
    let z0 = add_num(2i32) * 2;
    let z1 = apply(add_num, 2);
    let z2 = apply(translate, 2);
    println!("{}, {}, {}", z0, z1, z2);
}
```

方法

Rust通过 `impl` 关键字在 `struct` 、 `enum` 或者 `trait` 对象上实现方法调用语法 (method call syntax)。 关联函数 (associated function) 的第一个参数通常为 `self` 参数，有3种变体：

- `self` ，允许实现者移动和修改对象，对应的闭包特性为 `FnOnce` 。
- `&self` ，既不允许实现者移动对象也不允许修改，对应的闭包特性为 `Fn` 。
- `&mut self` ，允许实现者修改对象但不允许移动，对应的闭包特性为 `FnMut` 。

不含 `self` 参数的关联函数称为静态方法 (static method)。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }

    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    // use associated function and method chaining
    println!("{}", Circle::new(0.0, 0.0, 2.0).area());
}
```

特性

特性与接口

为了描述类型可以实现的抽象接口 (abstract interface)， Rust引入了特性 (trait) 来定义函数类型签名 (function type signature)：

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

其中函数 `print_area()` 中的泛型参数 `T` 被添加了一个名为 `HasArea` 的特性约束 (trait constraint)，用以确保任何实现了 `HasArea` 的类型将拥有一个 `.area()` 方法。如果需要多个特性限定 (multiple trait bounds)，可以使用 `+` ：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug
{
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

其中第二个例子使用了更灵活的 `where` 从句，它还允许限定的左侧可以是任意类型， 而不仅仅是类型参数。

定义在特性中的方法称为默认方法 (default method)，可以被该特性的实现覆盖。此外，特性之间也可以存在继承 (inheritance)：

```
trait Foo {
    fn foo(&self);

    // default method
    fn bar(&self) { println!("We called bar."); }
}

// inheritance
trait FooBar : Foo {
    fn foobar(&self);
}

struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

如果两个不同特性的方法具有相同的名称，可以使用通用函数调用语法 (universal function call syntax)：

```
// short-hand form
Trait::method(args);

// expanded form
<Type as Trait>::method(args);
```

关于实现特性的几条限制：

- 如果一个特性不在当前作用域内，它就不能被实现。
- 不管是特性还是 `impl`，都只能在当前的包装箱内起作用。
- 带有特性约束的泛型函数使用单态化实现 (monomorphization)，所以它是静态派分的 (statically dispatched)。

下面列举几个非常有用的标准库特性：

- `Drop` 提供了当一个值退出作用域后执行代码的功能，它只有一个 `drop(&mut self)` 方法。
- `Borrow` 用于创建一个数据结构时把拥有和借用的值看作等同。
- `AsRef` 用于在泛型中把一个值转换为引用。
- `Deref<Target=T>` 用于把 `&U` 类型的值自动转换为 `&T` 类型。
- `Iterator` 用于在集合 (collection) 和惰性值生成器 (lazy value generator) 上实现迭代器。
- `Sized` 用于标记运行时长度固定的类型，而不定长的切片和特性必须放在指针后面使其运行时长度已知， 比如 `&[T]` 和 `Box<Trait>` 。

泛型和多态

泛型 (generics) 在类型理论中称作参数多态 (parametric polymorphism)， 意为对于给定参数可以有多种形式的函数或类型。先看Rust中的一个泛型例子：

Option在rust标准库中的定义:

```
enum Option<T> {
    Some(T),
    None,
}
```

Option的典型用法:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

其中 `<T>` 部分表明它是一个泛型数据类型。当然，泛型参数也可以用于函数参数和结构体域：

```
// generic functions
fn make_pair<T, U>(a: T, b: U) -> (T, U) {
    (a, b)
}
let couple = make_pair("man", "female");

// generic structs
struct Point<T> {
    x: T,
    y: T,
}
let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

对于多态函数，存在两种派分 (dispatch) 机制：静态派分和动态派分。 前者类似于C++的模板，Rust会生成适用于指定类型的特殊函数，然后在被调用的位置进行替换， 好处是允许函数被内联调用，运行比较快，但是会导致代码膨胀 (code bloat)； 后者类似于Java或Go的 `interface` ，Rust通过引入特性对象 (trait object) 来实现， 在运行期查找虚表 (vtable) 来选择执行的方法。特性对象 `&Foo` 具有和特性 `Foo` 相同的名称， 通过转换 (casting) 或者强制多态化 (coercing) 一个指向具体类型的指针来创建。

当然，特性也可以接受泛型参数。但是，往往更好的处理方式是使用关联类型 (associated type)：

```
// use generic parameters
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
}

fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N)
 -> u32 {

}

// use associated types
trait Graph {
```

```
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}

fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> ui
nt {

}

struct Node;

struct Edge;

struct SimpleGraph;

impl Graph for SimpleGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {

    }

    fn edges(&self, n: &Node) -> Vec<Edge> {

    }
}

let graph = SimpleGraph;
let object = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

注释与文档

注释

在 Rust 里面注释分成两种，行注释和块注释。它的形式和 C 语言是一样的。 两种注释分别是：

> 1. 行注释使用 `//` 放在注释前面。比如:

```
// I love Rust, but I hate Rustc.
```

> 1. 块注释分别使用`/*`和`*/`包裹需要注释的内容。比如：

```
/* W-Cat 是个大胖猫，N-Cat 是个高度近视猫。*/
```

文档

Rust 自带有文档功能的注释，分别是`///`和`//!`。支持 Markdown 格式

1. `///`用来描述的它后面接着的项。
2. `//!`用来描述包含它的项，一般用在模块文件的头部。 比如在 main.rs 文件中输入以下内容：

```
        //! # The first line
        //! The second line
        /// Adds one to the number given.
        ///
        /// # Examples
        ///
        ///
```

````
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
````

```

## 生成 html 文档

- `rustdoc main.rs`

或者

- `cargo doc`
```

输入输出流

输入输出是人机交互的一种方式。最常见的输入输出是标准输入输出和文件输入输出（当然还有数据库输入输出，本节不讨论这部分）。

标准输入

标准输入也叫作控制台输入，是常见输入的一种。

例子**1**：

```
use std::io;

fn read_input() -> io::Result<()> {
    let mut input = String::new();

    try!(io::stdin().read_line(&mut input));

    println!("You typed: {}", input.trim());

    Ok(())
}

fn main() {
    read_input();
}
```

例子**2**：

```
use std::io;
fn main() {
    let mut input = String::new();

    io::stdin().read_line(&mut input).expect("WTF!");

    println!("You typed: {}", input.trim());
}
```

这里体现了常见的标准输入的处理方式。两个例子都是声明了一个可变的字符串来保存输入的数据。 他们的不同之处在在于处理潜在输入异常的方式。

1. 例子 1 使用了 `try!` 宏。这个宏会返回 `Result<(), io::Error>` 类型， `io::Result<()>` 就是这个类型的别名。所以例子 1 需要单独使用一个 `read_input` 函数来接收这个类型，而不是在 `main` 函数里面，因为 `main` 函数并没有接收 `io::Result<()>` 作为返回类型。
2. 例子 2 使用了 `Result<(), io::Error>` 类型的 `expect` 方法来接收 `io::stdin().read_line` 的返回类型。并处理可能潜在的 io 异常。

标准输出

标准输出也叫控制台输出，Rust 里面常见的标准输出宏有 `print!` 和 `println!` 。它们的区别是后者比前者在末尾多输出一个换行符。

例子**1**：

```
fn main() {
    print!("this ");
    print!("will ");
    print!("be ");
    print!("on ");
    print!("the ");
    print!("same ");
    print!("line ");

    print!("this string has a newline, why not choose println! instead?\n");
}
```

例子**2**：

```
fn main() {
    println!("hello there!");
    println!("format {} arguments", "some");
}
```

这里两个例子都比较简单。读者可以运行一下查看输出结果对比一下他们的区别。值得注意的是例子 2 中，`{ }` 会被 `"some"` 所替换。这是 rust 里面的一种格式化输出。

标准化的输出是行缓冲(line-buffered)的,这就导致标准化的输出在遇到一个新行之前并不会被隐式刷新。 换句话说 `print!` 和 `println!` 二者的效果并不总是相同的。 如果说得更简单明了一点就是，您不能把 `print!` 当做是C语言中的 `printf` 譬如：

```
use std::io;
fn main() {
    print!("请输入一个字符串：");
    let mut input = String::new();
    io::stdin()
        .read_line(&mut input)
        .expect("读取失败");
    print!("您输入的字符串是：{}\n", input);
}
```

在这段代码运行时则不会先出现预期的提示字符串，因为行没有被刷新。 如果想要达到预期的效果就要显示的刷新：

```
use std::io::{self, Write};
fn main() {
    print!("请输入一个字符串：");
    io::stdout().flush().unwrap();
    let mut input = String::new();
    io::stdin()
        .read_line(&mut input)
        .expect("读取失败");
    print!("您输入的字符串是：{}\n", input);
}
```

文件输入

文件输入和标准输入都差不多，除了输入流指向了文件而不是控制台。下面例子采用了模式匹配来处理潜在的输入错误

例子：

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main() {
    // 创建一个文件路径
    let path = Path::new("hello.txt");
    let display = path.display();

    // 打开文件只读模式, 返回一个 `io::Result<File>` 类型
    let mut file = match File::open(&path) {
        // 处理打开文件可能潜在的错误
        Err(why) => panic!("couldn't open {}: {}", display,
                                                   Error::descri
ption(&why)),
        Ok(file) => file,
    };

    // 文件输入数据到字符串，并返回 `io::Result<usize>` 类型
    let mut s = String::new();
    match file.read_to_string(&mut s) {
        Err(why) => panic!("couldn't read {}: {}", display,
                                                   Error::descri
ption(&why)),
        Ok(_) => print!("{} contains:\n{}", display, s),
    }
}
```

文件输出

文件输出和标准库输出也差不多，只不过是把输出流重定向到文件中。下面详细看例子。

例子：

```
// 输出文本
static LOREM_IPSUM: &'static str =
```

```
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed d
o eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad m
inim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate v
elit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.
";

use std::error::Error;
use std::io::prelude::*;
use std::fs::File;
use std::path::Path;

fn main() {
    let path = Path::new("out/lorem_ipsum.txt");
    let display = path.display();

    // 用只写模式打开一个文件，并返回 `io::Result<File>` 类型
    let mut file = match File::create(&path) {
        Err(why) => panic!("couldn't create {}: {}",
                           display,
                           Error::description(&why)),
        Ok(file) => file,
    };

    // 写入 `LOREM_IPSUM` 字符串到文件中, 并返回 `io::Result<()>` 类型

    match file.write_all(LOREM_IPSUM.as_bytes()) {
        Err(why) => {
            panic!("couldn't write to {}: {}", display,
                                               Error::descriptio
n(&why))
        },
        Ok(_) => println!("successfully wrote to {}", display),
```

```
        }
    }
```

cargo简介

曾几何时，对于使用惯了 `C/C++` 语言的猿们来说，项目代码的组织与管理绝对是一场噩梦。为了解决 `C/C++` 项目的管理问题，猿神们想尽了各种办法，开发出了各种五花八门的项目管理工具，从一开始的 `automake` 到后来的 `cmake` 、 `qmake` 等等，但结果并不如人意，往往是解决了一些问题，却引入了更多的问题， `C/C++` 猿们经常会陷入在掌握语言本身的同时，还要掌握复杂的构建工具语法的窘境。无独有偶， `java` 的项目代码组织与管理工具 `ant` 和 `maven` 也存在同样的问题。复杂的项目管理配置参数，往往让猿们不知所措。

作为一门现代语言， `rust` 自然要摒弃石器时代项目代码管理的方法和手段。 `rust` 项目组为各位猿提供了超级大杀器 `cargo` ，以解决项目代码管理所带来的干扰和困惑。用过 `node.js` 的猿们，应该对 `node.js` 中的神器 `npm` 、 `grunt` 、 `gulp` 等工具印象深刻。作为新一代静态语言中的翘楚， `rust` 官方参考了现有语言管理工具的优点，于是就产生了 `cargo` 。

言而总之，作为 `rust` 的代码组织管理工具， `cargo` 提供了一系列的工具，从项目的建立、构建到测试、运行直至部署，为 `rust` 项目的管理提供尽可能完整的手段。同时，与 `rust` 语言及其编译器 `rustc` 本身的各种特性紧密结合，可以说既是语言本身的知心爱人，又是 `rust` 猿们的贴心小棉袄，谁用谁知道。 废话就不多说了，直接上例子和各种高清无马图。

cargo入门

首先，当然还是废话，要使用cargo，自然首先要安装cargo。安装cargo有三种方法，前两种方法请参见rust的安装方法，因为cargo工具是官方正统出身，当然包含在官方的分发包中。第三种方法即从 `cargo` 项目的源码仓库进行构建。Oh，My God。的确是废话。

好了，假设各位已经安装好了cargo，大家和我一起学一下起手式。当然了，猿的世界，起手式一般都千篇一律——那就是 `hello world` 大法。 在终端中输入

```
$ cargo new hello_world --bin
```

上述命令使用**cargo new**在当前目录下新建了基于cargo项目管理的rust项目，项目名称为hello_world，--bin表示该项目将生成可执行文件。具体生成的项目目录结构如下：

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

大家可以在终端中输入上述命令，敲出回车键之后即可看到上述结果，或者直接去编辑器或文件管理器中去观察即可。 打开main.rs文件，可以看到，cargo new命令为我们自动生成了hello_world运行所必须的所有代码：

```
fn main() {
    println!("Hello, world!");
}
```

好了，心急的猿们可能已经迫不及待的脱裤子了，好吧，我们先来构建并看看cargo有多神奇，在终端中输入：

```
$ cargo build
```

稍等片刻，cargo会自动为我们构建好高清应用所需的一切，对于这个起手式来说，缓冲不会超过5秒，12秒88的选手要憋住了。

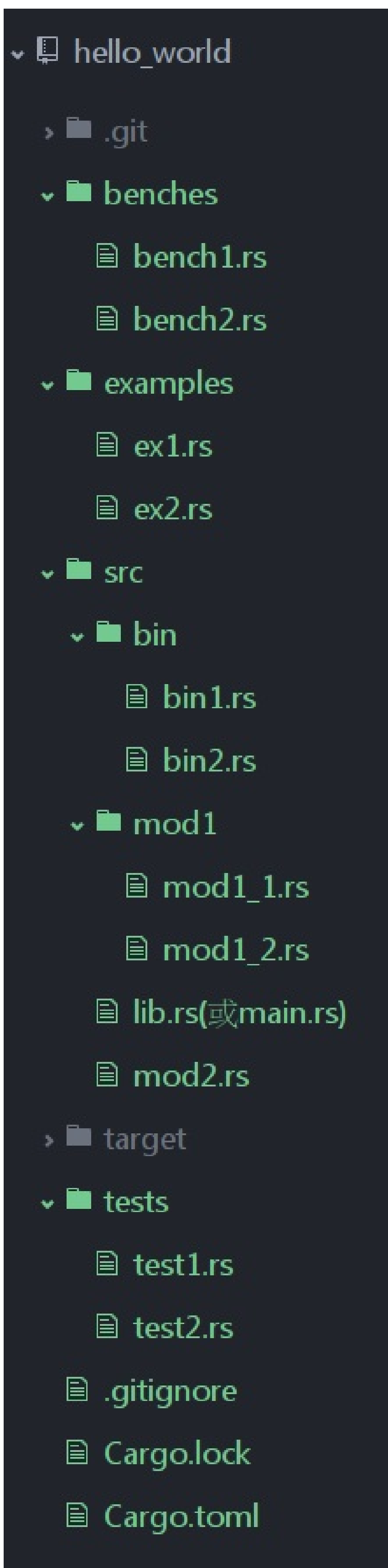
```
$ cargo run
    Running `target/debug/hello_world`
Hello, world!
```

看到了什么，看到了什么，吓尿了有木有，吓尿了有木有。好了，cargo就是这么简单。

当然了，说cargo美，并不仅仅是简单这么简单，cargo虽然简单，但是很强大。有多么强大？？可以说，基本上rust开发管理中所需的手段，cargo都有。很小很强大，既强又有节操，不带马，学习曲线几乎为零。

基于cargo的rust项目组织结构

这次不说废话了，先上高清无马图：



对上述cargo默认的项目结构解释如下：

`cargo.toml` 和 `cargo.lock` 文件总是位于项目根目录下。

源代码位于 `src` 目录下。

默认的库入口文件是 `src/lib.rs` 。

默认的可执行程序入口文件是 `src/main.rs` 。

其他可选的可执行文件位于 `src/bin/*.rs` (这里每一个rs文件均对应一个可执行文件)。

外部测试源代码文件位于 `tests` 目录下。

示例程序源代码文件位于 `examples` 。

基准测试源代码文件位于 `benches` 目录下。

好了，大家一定谨记这些默认规则，最好按照这种模式来组织自己的rust项目。

cargo.toml和cargo.lock

`cargo.toml` 和 `cargo.lock` 是cargo项目代码管理的核心两个文件，cargo工具的所有活动均基于这两个文件。

`cargo.toml` 是cargo特有的项目数据描述文件，对于猿们而言， `cargo.toml` 文件存储了项目的所有信息，它直接面向rust猿，猿们如果想让自己的rust项目能够按照期望的方式进行构建、测试和运行，那么，必须按照合理的方式构建'cargo.toml'。

而 `cargo.lock` 文件则不直接面向猿，猿们也不需要直接去修改这个文件。lock文件是cargo工具根据同一项目的toml文件生成的项目依赖详细清单文件，所以我们一般不用不管他，只需要对着 `cargo.toml` 文件撸就行了。

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["fuying"]

[dependencies]
```

toml文件是由诸如[package]或[dependencies]这样的段落组成，每一个段落又由多个字段组成，这些段落和字段就描述了项目组织的基本信息，例如上述toml文件中的[package]段落描述了 `hello_world` 项目本身的一些信息，包括项目名称（对应于name字段）、项目版本（对应于version字段）、作者列表（对应于authors字段）等；[dependencies]段落描述了 `hello_world` 项目的依赖项目有哪些。

下面我们来看看toml描述文件中常用段落和字段的意义。

package段落

[package]段落描述了软件开发者对本项目的各种元数据描述信息，例如[name]字段定义了项目的名称，[version]字段定义了项目的当前版本，[authors]定义了该项目的所有作者，当然，[package]段落不仅仅包含这些字段，[package]段落的其他可选字段详见cargo参数配置章节。

定义项目依赖

使用cargo工具的最大优势就在于，能够对该项目的各种依赖项进行方便、统一和灵活的管理。这也是使用cargo对rust 的项目进行管理的重要目标之一。在cargo的toml文件描述中，主要通过各种依赖段落来描述该项目的各种依赖项。toml中常用的依赖段落包括一下几种：

- 基于rust官方仓库crates.io，通过版本说明来描述：
- 基于项目源代码的git仓库地址，通过URL来描述：
- 基于本地项目的绝对路径或者相对路径，通过类Unix模式的路径来描述： 这三种形式具体写法如下：

```
[dependencies]
typemap = "0.3"
plugin = "0.2*"
hammer = { version = "0.5.0"}
color = { git = "https://github.com/bjz/color-rs" }
geometry = { path = "crates/geometry" }
```

上述例子中，2-4行为方法一的写法，第5行为方法二的写法，第6行为方法三的写法。 这三种写法各有用处，如果项目需要使用crates.io官方仓库来管理项目依赖项，推荐使用第一种方法。如果项目开发者更倾向于使用git仓库中最新的源码，可以使用方法二。方法二也经常用于当官方仓库的依赖项编译不通过时的备选方案。方法三主要用于源代码位于本地的依赖项。

定义集成测试用例

cargo另一个重要的功能，即将软件开发过程中必要且非常重要的测试环节进行集成，并通过代码属性声明或者toml文件描述来对测试进行管理。其中，单元测试主要通过在项目代码的测试代码部分前用 `#[test]` 属性来描述，而集成测试，则一般都会通过toml文件中的[[test]]段落进行描述。 例如，假设集成测试文件均位于tests文件夹下，则toml可以这样来写：

```
[[test]]
name = "testinit"
path = "tests/testinit.rs"

[[test]]
name = "testtime"
path = "tests/testtime.rs"
```

上述例子中，name字段定义了集成测试的名称，path字段定义了集成测试文件相对于本toml文件的路径。 看看，定义集成测试就是如此简单。 需要注意的是:

- 如果没有在Cargo.toml里定义集成测试的入口，那么tests目录(不包括子目录)下的每个rs文件被当作集成测试入口.
- 如果在Cargo.toml里定义了集成测试入口，那么定义的那些rs就是入口，不再默认指定任何集成测试入口.

定义项目示例和可执行程序

上面我们介绍了**cargo**项目管理中常用的三个功能，还有两个经常使用的功能：**example**用例的描述以及**bin**用例的描述。其描述方法和**test**用例描述方法类似。不过，这时候段落名称**'[[test]]'**分别替换为：**'[[example]]'**或者**'[[bin]]'**。例如：

```
[[example]]
name = "timeout"
path = "examples/timeout.rs"

[[bin]]
name = "bin1"
path = "bin/bin1.rs"
```

对于'[[example]]'和'[[bin]]'段落中声明的examples和bins，需要通过'cargo run --example NAME'或者'cargo run --bin NAME'来运行，其中NAME对应于你在name字段中定义的名称。

构建、清理、更新以及安装

领会了toml描述文件的写法，是一个重要的方面。另一个重要的方面，就是cargo工具本身为我们程序猿提供的各种好用的工具。如果大家感兴趣，自己在终端中输入'cargo --help'查看即可。其中开发时最常用的命令就是'cargo build'，用于构建项目。此外，'cargo clean'命令可以清理target文件夹中的所有内容；'cargo update'根据toml描述文件重新检索并更新各种依赖项的信息，并写入lock文件，例如依赖项版本的更新变化等等；'cargo install'可用于实际的生产部署。这些命令在实际的开发部署中均是非常有用的。

cargo更多详细用法请参见'28. cargo参数配置'

基本程序结构

Rust 是多范式语言，当然支持命令式编程风格。本章讲解 Rust 中的几种基本程序结构。

注释

Rust 代码文件中，通常我们可以看到 3 种注释。

- 行注释
- 文档注释
- 模块注释

行注释

`//` 后的，直到行尾，都属于注释，不会影响程序的行为。

```
// 创建一个绑定
let x = 5;

let y = 6; // 创建另一个绑定
```

文档注释

文档注释使用 `///` ，一般用于函数或结构体（字段）的说明，置于要说明的对象上方。文档注释内部可使用markdown格式的标记语法，可用于 rustdoc 工具的自动文档提取。

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

模块注释

模块注释使用 `//!` ，用于说明本模块的功能。一般置于模块文件的头部。

```
//! # The Rust Standard Library
//!
//! The Rust Standard Library provides the essential runtime
//! functionality for building portable Rust software.
```

PS: 相对于 `///` , `//!` 用来注释包含它的项（也就是说，crate，模块或者函数），而不是位于它之后的项。

其它：兼容C语言的注释

Rust 也支持兼容 C 的块注释写法： `/* */` 。但是不推荐使用，请尽量不要使用这种注释风格（会被鄙视的）。

```
/*
    let x = 42;
    println!("{}", x);
*/
```

条件分支

- if
- if let
- match

if 表达式

Rust 中的 if 表达式基本就是如下几种形式：

```
// 形式 1
if expr1 {

}

// 形式 2
if expr1 {

}
else {

}

// 形式 3
if expr1 {

}
else if expr2 {
    // else if 可多重
}
else {

}
```

相对于 C 系语言，Rust 的 if 表达式的显著特点是：

1. 判断条件不用小括号括起来；
2. 它是表达式，而不是语句。

鉴于上述第二点，因为是表达式，所以我们可以写出如下代码：

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

或者压缩成一行：

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

if let

我们在代码中常常会看到 `if let` 成对出现，这实际上是一个 match 的简化用法。直接举例来说明：

```
let x = Some(5);

if let Some(y) = x {
    println!("{}", y);      // 这里输出为：5
}

let z = if let Some(y) = x {
    y
}
else {
    0
};
// z 值为 5
```

上面代码等价于

```
let x = Some(5);
match x {
    Some(y) => println!("{}", y),
    None => ()
}

let z = match x {
    Some(y) => y,
    None => 0
};
```

设计这个特性的目的是，在条件判断的时候，直接做一次模式匹配，方便代码书写，使代码更紧凑。

match

Rust 中没有类似于 C 的 `switch` 关键字，但它有用于模式匹配的 `match`，能实现同样的功能，并且强大太多。

match 的使用非常简单，举例如下：

```
let x = 5;

match x {
    1 => {
        println!("one")
    },
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

注意，match 也是一个表达式。match 后面会专门论述，请参见 模式匹配 这一章。

循环

- for
- while
- loop
- break 与 continue
- label

for

for 语句用于遍历一个迭代器。

```
for var in iterator {
    code
}
```

Rust 迭代器返回一系列的元素，每个元素是循环中的一次重复。然后它的值与 var 绑定，它在循环体中有效。每当循环体执行完后，我们从迭代器中取出下一个值，然后我们再重复一遍。当迭代器中不再有值时，for 循环结束。

比如：

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

输出

```
0
1
2
3
4
5
6
7
8
9
```

不熟悉迭代器概念的同学可能傻眼了，下面不妨用 C 形式的 for 语句做下对比：

```
// C 语言的 for 循环例子
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

两者输出是相同的，那么，为何 Rust 要这样来设计 for 语句呢？

1. 简化边界条件的确定，减少出错；
2. 减少运行时边界检查，提高性能。

即使对于有经验的 C 语言开发者来说，要手动控制要循环的每个元素也都是复杂并且易于出错的。

for 语句就是迭代器遍历的语法糖。

上述迭代器的形式虽好，但是好像在循环过程中，少了索引信息。Rust 考虑到了这一点，当你需要记录你已经循环了多少次了的时候，你可以使用 `.enumerate()` 函数。比如：

```
for (i,j) in (5..10).enumerate() {
    println!("i = {} and j = {}", i, j);
}
```

输出：

```
i = 0 and j = 5
i = 1 and j = 6
i = 2 and j = 7
i = 3 and j = 8
i = 4 and j = 9
```

再比如：

```
let lines = "Content of line one
Content of line two
Content of line three
Content of line four".lines();
for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

输出：

```
0: Content of line one
1: Content of line two
2: Content of line three
3: Content of line four
```

关于迭代器的知识，详见 迭代器 章节。

while

Rust 提供了 while 语句，条件表达式为真时，执行语句体。当你不确定应该循环多少次时可选择 while。

```
while expression {
    code
}
```

比如：

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

loop

有一种情况，我们经常会遇到，就是写一个无限循环：

```
while true {
    // do something
}
```

针对这种情况，Rust 专门优化提供了一个语句 loop。

```
loop {
    // do something
}
```

`loop` 与 `while true` 的主要区别在编译阶段的静态分析。

比如说，如下代码：

```
let mut a;
loop {
     a = 1;
     // ... break ...
}
do_something(a)
```

如果是 `loop` 循环，编译器会正确分析出变量 `a` 会被正确初始化，而如果换成 `while true` ，则会发生编译错误。这个微小的区别也会影响生命周期分析。

break 和 continue

与 C 语言类似，Rust 也提供了 break 和 continue 两个关键字用来控制循环的流程。

- break 用来跳出当前层的循环；
- continue 用来执行当前层的下一次迭代。

像上面那个 while 例子：

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

可以优化成：

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

这样感觉更直观一点。

下面这个例子演示 continue 的用法：

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

    println!("{}", x);
}
```

它的作用是打印出 `0~9` 的奇数。结果如下：

```
1
3
5
7
9
```

label

你也许会遇到这样的情形，当你有嵌套的循环而希望指定你的哪一个 break 或 continue 该起作用。就像大多数语言，默认 break 或 continue 将会作用于当前层的循环。当你想要一个 break 或 continue 作用于一个外层循环，你可以使用标签来指定你的 break 或 continue 语句作用的循环。

如下代码只会在 x 和 y 都为奇数时打印他们：

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // continues the loop
 over x
        if y % 2 == 0 { continue 'inner; } // continues the loop
 over y
        println!("x: {}, y: {}", x, y);
    }
}
```

类型、运算符和字符串

本章讲解 Rust 中的类型相关基础知识、运算符相关知识、和字符串的基本知识。

原生类型

像其他现代编程语言一样，Rust提供了一系列基础的类型，我们一般称之为原生类型。其强大的类型系统就是建立在这些原生类型之上的，因此，在写Rust代码之前，必须要对Rust的原生类型有一定的了解。

bool

Rust自带了 `bool` 类型，其可能值为 `true` 或者 `false` 。我们可以通过这样的方式去声明它：

```
let is_she_love_me = false;
let mut is_he_love_me: bool = true;
```

当然，bool类型被用的最多的地方就是在 `if表达式` 里了。

char

在Rust中，一个 `char` 类型表示一个*Unicode*字符,这也就意味着，在某些语言里代表一个字符(8bit)的char，在Rust里实际上是四个字节(32bit)。同时，我们可以将各种奇怪的非中文字符随心所欲的赋值给一个char类型。需要注意的是，Rust中我们要用 `'` 来表示一个char，如果用 `"` 的话你得到的实际上是一个 `&'static str` 。

```
let c = 'x';
let cc = '王';
```

数字类型

和其他类C系的语言不一样，Rust用一种符号+位数的方式来表示其基本的数字类型。可能你习惯了 `int` 、 `double` 、 `float` 之类的表示法，Rust的表示法需要你稍微适应一下。

你可用的符号有 `i` 、 `f` 、 `u`

你可用的位数，当然了，都是2的n次幂，分别为 `8` 、 `16` 、 `32` 、 `64` 及 `size` 。

你可以将其组合起来，形成诸如 `i32` , `u16` 等类型。

当然了，这样的组合并不自由，因为浮点类型最少只能用32位来表示，因此只能有 `f32` 和 `f64` 来表示。

自适应类型

看完上面你一定会对 `isize` 和 `usize` 很好奇。这两个是来干啥的。这两个嘛，其实是取决于你的操作系统的位数。简单粗暴一点比如64位电脑上就是64位，32位电脑上就是32位，16位......呵呵哒。

但是需要注意的是，你不能因为你的电脑是64位的，而强行将它等同于64，也就是说 `isize != i64` ，任何情况下你都需要强制转换。

数组 array

Rust的数组是被表示为 `[T;N]` 。其中N表示数组大小，并且这个大小一定是个编译时就能获得的整数值，T表示 `泛型` 类型，即任意类型。我们可以这么来声明和使用一个数组:

```
let a = [8, 9, 10];
let b: [u8;3] = [8, 6, 5];
print!("{}", a[0]);
```

和Golang一样，Rust的数组中的 `N` （大小）也是类型的一部分，即 `[u8; 3] != [u8; 4]` 。这么设计是为了更安全和高效的使用内存，当然了，这会给第一次接触类似概念的人带来一点点困难，比如以下代码。

```
fn show(arr: [u8;3]) {
    for i in &arr {
        print!("{} ", i);
    }
}

fn main() {
    let a: [u8; 3] = [1, 2, 3];
    show(a);
    let b: [u8; 4] = [1, 2, 3, 4];
    show(b);
}
```

编译运行它你将获得一个编译错误：

```
<anon>:11:10: 11:11 error: mismatched types:
 expected `[u8; 3]`,
    found `[u8; 4]`
(expected an array with a fixed size of 3 elements,
    found one with 4 elements) [E0308]
<anon>:11     show(b);
                   ^
<anon>:11:10: 11:11 help: see the detailed explanation for E0308
error: aborting due to previous error
```

这是因为你将一个4长度的数组赋值给了一个只需要3长度数组作为参数的函数。那么如何写一个通用的show方法来展现任意长度数组呢？请看下节 `Slice`

Slice

`Slice` 从直观上讲，是对一个 `Array` 的切片，通过 `Slice` ，你能获取到一个 `Array` 的部分或者全部的访问权限。和 `Array` 不同， `Slice` 是可以动态的，但是呢，其范围是不能超过 `Array` 的大小，这点和Golang是不一样的。

一个 `Slice` 的表达式可以为如下: `&[T]` 或者 `&mut [T]` 。

这里 `&` 符号是一个难点，我们不妨放开这个符号，简单的把它看成是 `Slice` 的甲鱼臀部——规定。另外，同样的， `Slice` 也是可以通过下标的方式访问其元素，下标也是从0开始的哟。 你可以这么声明并使用一个 `Slice` ：

```
let arr = [1, 2, 3, 4, 5, 6];
let slice_complete = &arr[..]; // 获取全部元素
let slice_middle = &arr[1..4]; // 获取中间元素，最后取得的Slice为 [2, 3, 4] 。切片遵循左闭右开原则。
let slice_right = &arr[1..]; // 最后获得的元素为[2, 3, 4, 5, 6]，长度为5。
let slice_left = &arr[..3]; // 最后获得的元素为[1, 2, 3]，长度为3。
```

怎么样，了解了吧。 那么接下来我们用 `Slice` 来改造一下上面的函数

```
fn show(arr: &[u8]) {
    for i in arr {
        print!("{} ", i);
    }
    println!("");
}

fn main() {
    let a: [u8; 3] = [1, 2, 3];
    let slice_a = &a[..];
    show(slice_a);
    let b: [u8; 4] = [1, 2, 3, 4];
    show(&b[..]);
}
```

输出

```
1 2 3
1 2 3 4
```

动态数组 Vec

熟悉C++ STL的同学可能对C++的vector很熟悉，同样的，Rust也提供了一个类似的东西。他叫 `Vec` 。

在基础类型里讲 `Vec` 貌似是不太合适的，但在实际应用中的应用比较广泛，所以说先粗略的介绍一下，在集合类型的章节会有详细讲述。

在Rust里， `Vec` 被表示为 `Vec<T>` ，其中T是一个泛型。

下面介绍几种典型的 `Vec` 的用法:

```
let mut v1: Vec<i32> = vec![1, 2, 3]; // 通过vec!宏来声明
let v2 = vec![0; 10]; // 声明一个初始长度为10的值全为0的动态数组
println!("{}", v1[0]); // 通过下标来访问数组元素

for i in &v1 {
    print!("{}", i); // &Vec<i32> 可以通过 Deref 转换成 &[i32]
}

println!("");

for i in &mut v1 {
    *i = *i+1;
    print!("{}", i); // 可变访问
}
```

输出结果：

```
1
123
234
```

最原生字符串 str

你可以用 `str` 来声明一个字符串，事实上，Rust中，所有用 `""` 包裹起来的都可以称为 `&str` (注意这个 `&` ,这是难点，不用管他，不是么？)，但是这个类型被单独用的情况很少，因此，我们将在下一节着重介绍字符串类型。

函数类型 Functions

函数同样的是一个类型，这里只给大家普及一些基本的概念，函数类型涉及到比较高阶的应用，希望大家能在后面的 `闭包` 章节仔细参读

下面是一个小例子

```
fn foo(x: i32) -> i32 { x+1 }

let x: fn(i32) -> i32 = foo;

assert_eq!(11, x(10));
```

复合类型

元组(Tuple)

在别的语言里，你可能听过元组这个词，它表示一个大小、类型固定的有序数据组。在 Rust 中，情况并没有什么本质上的不同。不过 Rust 为我们提供了一系列简单便利的语法来让我们能更好的使用他。

```
let y = (2, "hello world");
let x: (i32, &str) = (3, "world hello");

// 然后呢，你能用很简单的方式去访问他们：

// 用 let 表达式
let (w, z) = y; // w=2, z="hello world"

// 用下标

let f = x.0; // f = 3
let e = x.1; // e = "world hello"
```

结构体(struct)

在Rust中，结构体是一个跟 `tuple` 类似 的概念。我们同样可以将一些常用的数据、属性聚合在一起，就形成了一个结构体。

所不同的是，Rust的结构体有三种最基本的形式。

具名结构体

这种结构体呢，他可以大致看成这样的一个声明形式:

```
struct A {
    attr1: i32,
    atrr2: String,
}
```

内部每个成员都有自己的名字和类型。

元组类型结构体

元组类型结构体使用小括号，类似 `tuple` 。

```
struct B(i32, u16, bool);
```

它可以看作是一个有名字的元组，具体使用方法和一般的元组基本类似。

空结构体

结构体内部也可以没有任何成员。

```
struct D;
```

空结构体的内存占用为0。但是我们依然可以针对这样的类型实现它的“成员函数”。

不过到目前为止，在 1.9 版本之前的版本，空结构体后面不能加大括号。 如果这么写，则会导致这部分的老编译器编译错误：

```
struct C {

}
```

实现结构体(impl)

Rust没有继承，它和Golang不约而同的选择了trait(Golang叫Interface)作为其实现多态的基础。可是，如果我们要想对一个结构体写一些专门的成员函数那应该怎么写呢？

答： impl

talk is cheap ,举个栗子：

```
struct Person {
    name: String,
}

impl Person {
    fn new(n: &str) -> Person {
        Person {
            name: n.to_string(),
        }
    }

    fn greeting(&self) {
        println!("{} say hello .", self.name);
    }
}

fn main() {
    let peter = Person::new("Peter");
    peter.greeting();
}
```

看见了 `self` ，Python程序员不厚道的笑了。

我们来分析一下，上面的 `impl` 中，new 被 Person 这个结构体自身所调用，其特征是 `::` 的调用，Java程序员站出来了：类函数！ 而带有 `self` 的 `greeting` ，更像是一个成员函数。

恩，回答正确，然而不加分。

关于各种ref的讨论

Rust 对代码有着严格的安全控制，因此对一个变量也就有了所有权和借用的概念。所有权同一时间只能一人持有，可变引用也只能同时被一个实例持有，不可变引用则可以被多个实例持有。同时所有权能被转移，在Rust中被称为 `move` 。

以上是所有权的基本概念，事实上，在整个软件的运行周期内，所有权的转换是一件极其恼人和烦琐的事情，尤其对那些初学 Rust 的同学来说。同样的，Rust 的结构体作为其类型系统的基石，也有着比较严格的所有权控制限制。具体来说，关于结构体的所有权，有两种你需要考虑的情况。

字段的 ref 和 owner

在以上的结构体中，我们定义了不少结构体，但是如你所见，结构体的每个字段都是完整的属于自己的。也就是说，每个字段的 owner 都是这个结构体。每个字段的生命周期最终都不会超过这个结构体。

但是有些时候，我只是想要持有一个(可变)引用的值怎么办？如下代码：

```
struct RefBoy {
    loc: &i32,
}
```

这时候你会得到一个编译错误：

```
<anon>:6:14: 6:19 error: missing lifetime specifier [E0106]
<anon>:6         loc: & i32,
```

这种时候，你将持有一个值的引用，因为它本身的生命周期在这个结构体之外，所以对这个结构体而言，它无法准确的判断获知这个引用的生命周期，这在 Rust 编译器而言是不被接受的。 因此，这个时候就需要我们给这个结构体人为的写上一个生命周期，并显式地表明这个引用的生命周期。写法如下：

```
struct RefBoy<'a> {
    loc: &'a i32,
}
```

这里解释一下这个符号 `<>` ，它表示的是一个 `属于` 的关系，无论其中描述的是 生命周期 还是 泛型 。即： `RefBoy in 'a` 。最终我们可以得出个结论， `RefBoy` 这个结构体，其生命周期一定不能比 `'a` 更长才行。

写到这里，可能有的人还是对生命周期比较迷糊，不明白其中缘由，其实你只需要知道两点即可：

1. 结构体里的引用字段必须要有显式的生命周期
2. 一个被显式写出生命周期的结构体，其自身的生命周期一定小于等于其显式写出的任意一个生命周期

关于第二点，其实生命周期是可以写多个的，用 `,` 分隔。

注：生命周期和泛型都写在 `<>` 里，先生命周期后泛型，用`,`分隔。

impl中的三种self

前面我们知道，Rust中，通过impl可以对一个结构体添加成员方法。同时我们也看到了 `self` 这样的关键字，同时，这个self也有好几种需要你仔细记忆的情况。

impl中的self,常见的有三种形式：`self` 、 `&self` 、`&mut self` ，我们分别来说。

被move的self

正如上面例子中的impl，我们实现了一个以 `self` 为第一个参数的函数，但是这样的函数实际上是有问题的。 问题在于Rust的所有权转移机制。

我曾经见过一个关于Rust的笑话："你调用了一下别人，然后你就不属于你了"。

比如下面代码就会报出一个错误：

```
struct A {
    a: i32,
}
impl A {
    pub fn show(self) {
        println!("{}", self.a);
    }
}

fn main() {
    let ast = A{a: 12i32};
    ast.show();
    println!("{}", ast.a);
}
```

错误：

```
13:25 error: use of moved value: `ast.a` [E0382]
<anon>:13     println!("{}", ast.a);
```

为什么呢？因为 Rust 本身，在你调用一个函数的时候，如果传入的不是一个引用，那么无疑，这个参数将被这个函数吃掉，即其 owner 将被 move 到这个函数的参数上。同理， `impl` 中的 `self` ，如果你写的不是一个引用的话，也是会被默认的 move 掉哟！

那么如何避免这种情况呢？答案是 `Copy` 和 `Clone` ：

```
#[derive(Copy, Clone)]
struct A {
    a: i32,
}
```

这么写的话，会使编译通过。但是这么写实际上也是有其缺陷的。其缺陷就是： `Copy` 或者 `Clone` ，都会带来一定的运行时开销！事实上，被move的 `self` 其实是相对少用的一种情况，更多的时候，我们需要的是 `ref` 和 `ref mut` 。

ref 和 ref mut

关于 `ref` 和 `mut ref` 的写法和被 move 的 `self` 写法类似，只不过多了一个引用修饰符号，上面有例子，不多说。

需要注意的一点是，你不能在一个 `&self` 的方法里调用一个 `&mut ref` ，任何情况下都不行！

但是，反过来是可以的。代码如下：

```
#[derive(Copy, Clone)]
struct A {
    a: i32,
}
impl A {
    pub fn show(&self) {
        println!("{}", self.a);
        // compile error: cannot borrow immutable borrowed conte
nt `*self` as mutable
        // self.add_one();
    }
    pub fn add_two(&mut self) {
        self.add_one();
        self.add_one();
        self.show();
    }
    pub fn add_one(&mut self) {
        self.a += 1;
    }
}

fn main() {
    let mut ast = A{a: 12i32};
    ast.show();
    ast.add_two();
}
```

需要注意的是，一旦你的结构体持有一个可变引用，你，只能在 `&mut self` 的实现里去改变他！

Rust允许我们灵活的对一个 struct 进行你想要的实现，在编程的自由度上无疑有了巨大的提高。

至于更高级的关于 trait 和泛型的用法，我们将在以后的章节进行详细介绍。

枚举类型 enum

Rust的枚举(`enum`)类型，跟C语言的枚举有点接近，然而更强大，事实上它是一种代数数据类型(Algebraic Data Type)。

比如说，这是一个代表东南西北四个方向的枚举：

```
enum Direction {
    West,
    North,
    Sourth,
    East,
}
```

但是，rust 的枚举能做到的，比 C 语言的更多。 比如，枚举里面居然能包含一些你需要的，特定的数据信息！ 这是常规的枚举所无法做到的，更像枚举类，不是么？

```
enum SpecialPoint {
    Point(i32, i32),
    Special(String),
}
```

你还可以给里面的字段命名，如

```
enum SpecialPoint {
    Point {
        x: i32,
        y: i32,
    },
    Special(String),
}
```

使用枚举

和struct的成员访问符号 `.` 不同的是，枚举类型要想访问其成员，几乎无一例外的要用到模式匹配。并且， 你可以写一个 `Direction::West` ，但是你现在还不能写成 `Direction.West` , 除非你显式的 `use` 它 。虽然编译器足够聪明能发现你这个粗心的毛病。

关于模式匹配，我不会说太多，还是举个栗子

```
enum SpecialPoint {
    Point(i32, i32),
    Special(String),
}

fn main() {
    let sp = SpecialPoint::Point(0, 0);
    match sp {
        SpecialPoint::Point(x, y) => {
            println!("I'am SpecialPoint(x={}, y={})", x, y);
        }
        SpecialPoint::Special(why) => {
            println!("I'am Special because I am {}", why);
        }
    }
}
```

呐呐呐，这就是模式匹配取值啦。 当然了， `enum` 其实也是可以 `impl` 的，一般人我不告诉他！

对于带有命名字段的枚举，模式匹配时可指定字段名

```
match sp {
    SpecialPoint::Point { x: x, y: y } => {
        // ...
    },
    SpecialPoint::Special(why) => {}
}
```

对于带有字段名的枚举类型，其模式匹配语法与匹配 `struct` 时一致。如

```
struct Point {
    x: i32,
    y: i32,
}

let point = Point { x: 1, y: 2 };

let Point { x: x, y: y } = point;
// 或
let Point { x, y } = point;
// 或
let Point { x: x, .. } = point;
```

模式匹配的语法与 `if let` 和 `let` 是一致的，所以在后面的内容中看到的也支持同样的语法。

String

这章我们来着重介绍一下字符串。

刚刚学习Rust的同学可能会被Rust的字符串搞混掉，比如 `str` ， `String` ， `OsStr` ， `CStr` ， `CString` 等等…… 事实上，如果你不做FFI的话，常用的字符串类型就只有前两种。我们就来着重研究一下Rust的前两种字符串。

你要明白的是，Rust中的字符串实际上是被编码成UTF-8的一个字节数组。这么说比较拗口，简单来说，Rust字符串内部存储的是一个u8数组，但是这个数组是Unicode字符经过UTF-8编码得来的。因此，可以看成Rust原生就支持Unicode字符集（Python2的码农泪流满面）。

str

首先我们先来看一下 `str` ， 从字面意思上，Rust的string被表达为： `&'static str` (看不懂这个表达式没关系，&表示引用你知道吧，static表示静态你知道吧，好了，齐了)，即，你在代码里写的，所有的用 `""` 包裹起来的字符串，都被声明成了一个不可变，静态的字符串。而我们的如下语句：

```
let x = "Hello";
let x:&'static str = "Hello";
```

实际上是将 `"Hello"` 这个静态变量的引用传递给了 `x` 。同时，这里的字符串不可变！

字符串也支持转义字符： 比如如下：

```
let z = "foo
bar";
let w = "foo\nbar";
assert_eq!(z, w);
```

也可以在字符串字面量前加上 `r` 来避免转义

```
//没有转义序列
let d: &'static str = r"abc \n abc";
//等价于
let c: &'static str = "abc \\n abc";
```

String

光有 `str` ，确实不够什么卵用，毕竟我们在实际应用中要的更多的还是一个可变的，不定长的字符串。这时候，一种在堆上声明的字符串 `String` 被设计了出来。它能动态的去增长或者缩减，那么怎么声明它呢？我们先介绍一种简单的方式，从 `str` 中转换：

```
let x:&'static str = "hello";

let mut y:String = x.to_string();
println!("{}", y);
y.push_str(", world");
println!("{}", y);
```

我知道你一定会问：—— 那么如何将一个 `String` 重新变成 `&str` 呢？ 答：用 `&*` 符号

```
fn use_str(s: &str) {
    println!("I am: {}", s);
}

fn main() {
    let s = "Hello".to_string();
    use_str(&*s);
}
```

我们来分析一下，以下部分将涉及到部分 `Deref` 的知识，可能需要你预习一下，如果不能理解大可跳过下一段：

首先呢， `&*` 是两个符号 `&` 和 `*` 的组合，按照Rust的运算顺序，先对 `String` 进行 `Deref` ,也就是 `*` 操作。

由于 `String` 实现了 `impl Deref<Target=str> for String` ，这相当于一个运算符重载，所以你就能通过 `*` 获得一个 `str` 类型。但是我们知道，单独的 `str` 是不能在Rust里直接存在的，因此，我们需要先给他进行 `&` 操作取得 `&str` 这个结果。

有人说了，我发现只要用 `&` 一个操作符就能将使上面的编译通过。 这其实是一个编译器的锅，因为Rust的编译器会在 `&` 后面插入足够多的 `*` 来尽可能满足 `Deref` 这个特性。这个特性会在某些情况下失效，因此，为了不给自己找麻烦，还是将操作符写全为好。

需要知道的是，将 `String` 转换成 `&str` 是非常轻松的，几乎没有任何开销。但是反过来，将 `&str` 转换成 `String` 是需要在堆上请求内存的，因此，要慎重。

我们还可以将一个UTF-8编码的字节数组转换成String，如

```
// 存储在Vec里的一些字节
let miao = vec![229,150,181];

// 我们知道这些字节是合法的UTF-8编码字符串，所以直接unwrap()
let meow = String::from_utf8(miao).unwrap();

assert_eq!("喵", meow);
```

索引访问

有人会把Rust中的字符串和其惯用的字符串等同起来，于是就出现了如下代码

```
let x = "hello".to_string();
x[1]; //编译错误！
```

Rust的字符串实际上是不支持通过下标访问的，但是呢，我们可以通过将其转变成数组的方式访问

```
let x = "哎哟我去".to_string();
for i in x.as_bytes() {
    print!("{} ", i);
}

println!("");

for i in x.chars() {
    print!("{}", i);
}

x.chars().nth(2);
```

字符串切片

对字符串切片是一件非常危险的事，虽然Rust支持，但是我并不推荐。因为Rust的字符串Slice实际上是切的bytes。这也就造成了一个严重后果，如果你切片的位置正好是一个Unicode字符的内部，Rust会发生Runtime的panic，导致整个程序崩溃。 因为这个操作是如此的危险，所以我就不演示了……

操作符和格式化字符串

现在的Rust资料，无论是Book还是RustByExample都没有统一而完全的介绍Rust的操作符。一个很重要的原因就是，Rust的操作符号和C++大部分都是一模一样的。

一元操作符

顾名思义，一元操作符是专门对一个Rust元素进行操纵的操作符，主要包括以下几个:

- `-` : 取负，专门用于数值类型。
- `*` : 解引用。这是一个很有用的符号，和 `Deref` （ `DerefMut` ）这个trait关联密切。
- `!` : 取反。取反操作相信大家都比较熟悉了，不多说了。有意思的是，当这个操作符对数字类型使用的时候，会将其每一位都置反！也就是说，你对一个 `1u8` 进行 `!` 的话你将会得到一个 `254u8` 。
- `&` 和 `&mut` : 租借，borrow。向一个owner租借其使用权，分别是租借一个只读使用权和读写使用权。

二元操作符

算数操作符

算数运算符都有对应的trait的，他们都在 `std::ops` 下：

- `+` : 加法。实现了 `std::ops::Add` 。
- `-` : 减法。实现了 `std::ops::Sub` 。
- `*` : 乘法。实现了 `std::ops::Mul` 。
- `/` : 除法。实现了 `std::ops::Div` 。
- `%` : 取余。实现了 `std::ops::Rem` 。

位运算符

和算数运算符差不多的是，位运算也有对应的trait。

- `&` : 与操作。实现了 `std::ops::BitAnd` 。
- `|` : 或操作。实现了 `std::ops::BitOr` 。
- `^` : 异或。实现了 `std::ops::BitXor` 。
- `<<` : 左移运算符。实现了 `std::ops::Shl` 。
- `>>` : 右移运算符。实现了 `std::ops::Shr` 。

惰性boolean运算符

逻辑运算符有三个，分别是 `&&` 、 `||` 、 `!` 。其中前两个叫做惰性boolean运算符，之所以叫这个名字。是因为在Rust里也会出现其他类C语言的逻辑短路问题。所以取了这么一个高大上然并卵的名字。 其作用和C语言里的一毛一样啊！哦，对了，有点不同的是Rust里这个运算符只能用在bool类型变量上。什么 `1 && 1` 之类的表达式给我死开。

比较运算符

比较运算符其实也是某些trait的语法糖啦，不同的是比较运算符所实现的trait只有两个 `std::cmp::PartialEq` 和 `std::cmp::PartialOrd`

其中， `==` 和 `!=` 实现的是 `PartialEq` 。 而， `<` 、 `>` 、 `>=` 、 `<=` 实现的是 `PartialOrd` 。

边看本节边翻开标准库（好习惯，鼓励）的同学一定会惊奇的发现，不对啊， `std::cmp` 这个mod下明明有四个trait，而且从肉眼上来看更符合逻辑的 `Ord` 和 `Eq` 岂不是更好？其实，Rust对于这四个trait的处理是很明确的。分歧主要存在于浮点类型。 熟悉IEEE的同学一定知道浮点数有一个特殊的值叫 `NaN` ，这个值表示未定义的一个浮点数。在Rust中可以用 `0.0f32 / 0.0f32` 来求得其值。那么问题来了，这个数他是一个确定的值，但是它表示的是一个不确定的数！那么 `NaN != NaN` 的结果是啥？标准告诉我们，是 `true` 。但是这么写又不符合 `Eq` 的定义里 `total equal` (每一位一样两个数就一样)的定义。因此有了 `PartialEq` 这么一个定义，我们只支持部分相等好吧，NaN这个情况我就给它特指了。

为了普适的情况，Rust的编译器选择了 `PartialOrd` 和 `PartialEq` 来作为其默认的比较符号的trait。我们也就和中央保持一致就好。

类型转换运算符

其实这个并不算运算符，因为他是个单词 `as` 。

这个就是C语言中各位熟悉的显式类型转换了。

show u the code:

```
fn avg(vals: &[f64]) -> f64 {
    let sum: f64 = sum(vals);
    let num: f64 = len(vals) as f64;
    sum / num
}
```

重载运算符

上面说了很多trait。有人会问了，你说这么多干啥？

答，为了运算符重载！

Rust是支持运算符重载的（某咖啡语言哭晕在厕所）。

关于这部分呢，在本书的第30节会有很详细的叙述，因此在这里我就不铺开讲了，上个栗子给大家，仅作参考：

```
use std::ops::{Add, Sub};

#[derive(Copy, Clone)]
struct A(i32);

impl Add for A {
    type Output = A;
    fn add(self, rhs: A) -> A {
        A(self.0 + rhs.0)
    }
}

impl Sub for A {
    type Output = A;
    fn sub(self, rhs: A) -> A{
        A(self.0 - rhs.0)
    }
}

fn main() {
    let a1 = A(10i32);
    let a2 = A(5i32);
    let a3 = a1 + a2;
    println!("{}", (a3).0);
    let a4 = a1 - a2;
    println!("{}", (a4).0);
}
```

output:

```
15
5
```

格式化字符串

说起格式化字符串，Rust采取了一种类似Python里面format的用法，其核心组成是五个宏和两个trait: `format!` 、 `format_arg!` 、 `print!` 、 `println!` 、 `write!` ; `Debug` 、 `Display` 。

相信你们在写Rust版本的Hello World的时候用到了 `print!` 或者 `println!` 这两个宏，但是其实最核心的是 `format!` ，前两个宏只不过将 `format!` 的结果输出到了console而已。

那么，我们来探究一下 `format!` 这个神奇的宏吧。

在这里呢，列举 `format!` 的定义是没卵用的，因为太复杂。我只为大家介绍几种典型用法。学会了基本上就能覆盖你平时80%的需求。

首先我们来分析一下format的一个典型调用

```
fn main() {
    let s = format!("{1}是个有着{0:>0width$}KG重，{height:?}cm高的
大胖子",
                    81, "wayslog", width=4, height=178);
    // 我被逼的牺牲了自己了……
    print!("{}", s);
}
```

我们可以看到， `format!` 宏调用的时候参数可以是任意类型，而且是可以position参数和key-value参数混合使用的。但是要注意的一点是，key-value的值只能出现在position值之后并且不占position。例如例子里你用 `3$` 引用到的绝对不是 `width` ，而是会报错。 这里面关于参数稍微有一个规则就是，参数类型必须要实现 `std::fmt` mod 下的某些trait。比如我们看到原生类型大部分都实现了 `Display` 和 `Debug` 这两个宏，其中整数类型还会额外实现一个 `Binary` ，等等。

当然了，我们可以通过 `{:type}` 的方式去调用这些参数。

比如这样：

```
format!("{:b}", 2);
// 调用 `Binary` trait
// Get : 10
format!("{:?}", "Hello");
// 调用 `Debug`
// Get : "Hello"
```

另外请记住：type这个地方为空的话默认调用的是 `Display` 这个trait。

关于 `:` 号后面的东西其实还有更多式子，我们从上面的 `{0:>0width$}` 来分析它。

首先 `>` 是一个语义，它表示的是生成的字符串向右对齐，于是我们得到了 `0081` 这个值。与之相对的还有 `<` (向左对齐)和 `^` (居中)。

再接下来 `0` 是一种特殊的填充语法，他表示用0补齐数字的空位，要注意的是，当0作用于负数的时候，比如上面例子中wayslog的体重是-81，那么你最终将得到 `-0081` ;当然了，什么都不写表示用空格填充啦;在这一位上，还会出现 `+` 、 `#` 的语法，使用比较诡异，一般情况下用不上。

最后是一个组合式子 `width$` ，这里呢，大家很快就能认出来是表示后面key-value值对中的 `width=4` 。你们没猜错，这个值表示格式化完成后字符串的长度。它可以是一个精确的长度数值，也可以是一个以 `$` 为结尾的字符串， `$` 前面的部分可以写一个key或者一个postion。

最后，你需要额外记住的是，在width和type之间会有一个叫精度的区域（可以省略不写如例子），他们的表示通常是以 `.` 开始的，比如 `.4` 表示小数点后四位精度。最让人遭心的是，你仍然可以在这个位置引用参数，只需要和上面width一样，用 `.N$` 来表示一个position的参数，但是就是不能引用key-value类型的。这一位有一个特殊用法，那就是 `.*` ，它不表示一个值，而是表示两个值！第一个值表示精确的位数，第二个值表示这个值本身。这是一种很尴尬的用法，而且极度容易匹配到其他参数。因此，我建议在各位能力或者时间不欠缺的时候尽量把格式化表达式用标准的形式写的清楚明白。尤其在面对一个复杂的格式化字符串的时候。

好了好了，说了这么多，估计你也头昏脑涨的了吧，下面来跟我写一下format宏的完整用法。仔细体会并提炼每一个词的意思和位置。

```
format_string := <text> [ format <text> ] *
format := '{' [ argument ] [ ':' format_spec ] '}'
argument := integer | identifier

format_spec := [[fill]align][sign]['#'][0][width]['.' precision]
[type]
fill := character
align := '<' | '^' | '>'
sign := '+' | '-'
width := count
precision := count | '*'
type := identifier | ''
count := parameter | integer
parameter := integer '$'
```

最后，留个作业吧。 给出参数列表如下： `(500.0, 12, "ELTON", "QB", 4, CaiNiao="Mike")`

请写出能最后输出一下句子并且将参数都被用过至少一遍的格式化字符串，并自己去play实验一下。

```
rust.cc社区的唐Mike眼睛度数足有0500.0度却还是每天辛苦码代码才能赚到100个Q
B。
但是ELTON却只需睡  12  个小时就可以迎娶白富美了。
```

函数

尽管rust是一门多范式的编程语言，但rust的编程风格是更偏向于函数式的，函数在rust中是“一等公民”——first-class type。这意味着，函数是可以作为数据在程序中进行传递，如：作为函数的参数。跟C、C++一样，rust程序也有一个唯一的程序入口-main函数。rust的main函数形式如下：

```
fn main() {
  //statements
}
```

rust使用 `fn` 关键字来声明和定义函数， `fn` 关键字隔一个空格后跟函数名，函数名后跟着一个括号，函数参数定义在括号内。rust使用 `snake_case` 风格来命名函数，即所有字母小写并使用下划线类分隔单词，如： `foo_bar` 。如果函数有返回值，则在括号后面加上箭头 **->** ，在箭头后加上返回值的类型。

这一章我们将学习以下与函数相关的知识：

1. 函数参数
2. 函数返回值
3. 语句和表达式
4. 高阶函数

> 注：本章所有例子均在**rustc1.4**下编译通过，且例子中说明的所有的编译错误都是**rustc1.4**版本给出的。

函数参数

参数声明

rust的函数参数声明和一般的变量声明相仿，也是参数名后加冒号，冒号后跟参数类型，不过不需要 `let` 关键字。需要注意的是，普通变量声明(let语句)是可以省略变量类型的，而函数参数的声明则不能省略参数类型。 来看一个简单例子：

```
fn main() {
  say_hi("ruster");
}

fn say_hi(name: &str) {
  println!("Hi, {}", name);
}
```

上例中， `say_hi` 函数拥有一个参数，名为 `name` ，类型为 `&str` 。

将函数作为参数

在rust中，函数是一等公民（可以储存在变量/数据结构中，可以作为参数传入函数，可以作为返回值），所以rust的函数参数不仅可以是一般的类型，也可以是函数。如：

```
fn main() {
  let xm = "xiaoming";
  let xh = "xiaohong";
  say_what(xm, hi);
  say_what(xh, hello);
}

fn hi(name: &str) {
  println!("Hi, {}.", name);
}

fn hello(name: &str) {
  println!("Hello, {}.", name);
}

fn say_what(name: &str, func: fn(&str)) {
  func(name)
}
```

上例中， `hi` 函数和 `hello` 函数都是只有一个 `&str` 类型的参数且没有返回值。而 `say_what` 函数则有两个参数，一个是 `&str` 类型，另一个则是函数类型（function type），它是只有一个 `&str` 类型参数且没有返回值的函数类型。

模式匹配

支持模式匹配，让rust平添了许多的灵活性，用起来也是十分的舒爽。模式匹配不仅可以用在变量声明（let语句）中，也可以用在函数参数声明中，如：

```
fn main() {
  let xm = ("xiaoming", 54);
  let xh = ("xiaohong", 66);
  print_id(xm);
  print_id(xh);
  print_name(xm);
  print_age(xh);
  print_name(xm);
  print_age(xh);
}

fn print_id((name, age): (&str, i32)) {
  println!("I'm {},age {}.", name, age);
}

fn print_age((_, age): (&str, i32)) {
  println!("My age is  {}", age);
}

fn print_name((name,_): (&str, i32)) {
  println!("I am  {}", name);
}
```

上例是一个元组(Tuple)匹配的例子，当然也可以是其他可在let语句中使用的类型。参数的模式匹配跟let语句的匹配一样，也可以使用下划线来表示丢弃一个值。

函数返回值

在rust中，任何函数都有返回类型，当函数返回时，会返回一个该类型的值。我们先来看看main函数：

```
fn main() {
  //statements
}
```

之前有说过，函数的返回值类型是在参数列表后，加上箭头和类型来指定的。不过，一般我们看到的main函数的定义并没有这么做。这是因为main函数的返回值是 `()` ，在rust中，当一个函数返回 `()` 时，可以省略。main函数的完整形式如下：

```
fn main() -> () {
  //statements
}
```

main函数的返回值类型是 `()` ，它是一个特殊的元组——没有元素的元组，称为 `unit` ，它表示一个函数没有任何信息需要返回。在Rust Reference的8.1.3 Tuple types中是的描述如下：

> For historical reasons and convenience, the tuple type with no elements (**()**) is often called ‘unit’ or ‘the unit type’.

`()` 类型，其实类似于C/C++、Java、C#中的 `void` 类型。

下面来看一个有返回值的例子：

```
fn main() {
  let a = 3;
  println!("{}", inc(a));
}

fn inc(n: i32) -> i32 {
  n + 1
}
```

上面的例子中，函数 `inc` 有一个 `i32` 类型的参数和返回值，作用是将参数加1返回。需要注意的是 `inc` 函数中只有 `n+1` 一个表达式，并没有像C/C++或Java、C#等语言有显式地 `return` 语句类返回一个值。这是因为，与其他基于语句的语言（如C语言）不同，rust是基于表达式的语言，函数中最后一个表达式的值，默认作为返回值。当然，rust中也有语句，关于rust的语句和表达式，请看下一节。

return关键字

rust也有 `return` 关键字，不过一般用于提前返回。来看一个简单地例子：

```
fn main() {
  let a = [1,3,2,5,9,8];
  println!("There is 7 in the array: {}", find(7, &a));
  println!("There is 8 in the array: {}", find(8, &a));
}

fn find(n: i32, a: &[i32]) -> bool {
  for i in a {
    if *i == n {
      return true;
    }
  }
  false
}
```

上例中， `find` 函数，接受一个 `i32` 类型 `n` 和一个 `i32` 类型的切片(`slice`) `a` ，返回一个 `bool` 值，若n是a的元素，则返回 `true` ，否则返回 `false` 。可以看到， `return` 关键字，用在 `for` 循环的 `if` 表达式中，若此

时a的元素与n相等，则立刻返回true，剩下的循环不必再进行，否则一直循环检测完整个切片(slice)，最后返回false。当然，return语句也可以用在最后返回，像C/C++一样使用：把 `find` 函数最后一句 `false` 改为 `return false;` （注意分号不可省略）也是可以的，不过这就不是rust的编程风格了。这里需要注意的是， `for` 循环中的 `i` ，其类型为 `&i32` ，需要使用解引用操作符来变换为 `i32` 类型。另外，切片（slice）在这里可以看作是对数组的引用，关于切片与数组的详细解释可以看Rust Reference和rustbyexample中的相关内容。

返回多个值

rust的函数不支持多返回值，但是我们可以利用元组来返回多个值，配合rust的模式匹配，使用起来十分灵活。先看例子：

```
fn main() {
  let (p2,p3) = pow_2_3(789);
  println!("pow 2 of 789 is {}.", p2);
  println!("pow 3 of 789 is {}.", p3);
}

fn pow_2_3(n: i32) -> (i32, i32) {
  (n*n, n*n*n)
}
```

可以看到，上例中， `pow_2_3` 函数接收一个 `i32` 类型的值，返回其二次方和三次方的值，这两个值包装在一个元组中返回。在 `main` 函数中， `let` 语句就可以使用模式匹配将函数返回的元组进行解构，将这两个返回值分别赋给 `p2` 和 `p3` ，从而可以得到 `789` 二次方的值和三次方的值。

发散函数

发散函数（diverging function）是rust中的一个特性。发散函数不返回，它使用感叹号 `!` 作为返回类型表示：

```
fn main() {
  println!("hello");
  diverging();
  println!("world");
}

fn diverging() -> ! {
  panic!("This function will never return");
}
```

由于发散函数不会返回，所以就算其后再有其他语句也是不会执行的。倘若其后还有其他语句，会出现如下编译警告：

```
<std macros>:2:1: 2:58 warning: unreachable statement, #[warn(unreachable_code)] on by default
<std macros>:2 $ crate:: io:: _print ( format_args ! ( $ ( $ arg ) * ) ) ) ;
               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<std macros>:1:23: 1:60 note: in this expansion of print! (defined in <std macros>)
a.rs:4:5: 4:23 note: in this expansion of println! (defined in <std macros>)
```

。当然了，我们要知道的是不发散的函数也是可以不返回的，比如无限循环之类的。 发散函数一般都以 `panic!` 宏调用或其他调用其他发散函数结束，所以，调用发散函数会导致当前线程崩溃。Rust Reference 6.1.3.2 Diverging functions中的描述如下：

> We call such functions "diverging" because they never return a value to the caller. Every control path in a diverging function must end with a panic!() or a call to another diverging function on every control path. The ! annotation does not denote a type.

语句和表达式

rust是一个基于表达式的语言，不过它也有语句。rust只有两种语句：声明语句和表达式语句，其他的都是表达式。基于表达式是函数式语言的一个重要特征，表达式总是返回值。

声明语句

rust的声明语句可以分为两种，一种为变量声明语句，另一种为Item声明语句。

1. 变量声明语句。主要是指 `let` 语句，如:

    ```
    let a = 8;
    let b: Vec<f64> = Vec::new();
    let (a, c) = ("hi", false);
    ```

    由于let是语句，所以不能将let语句赋给其他值。如下形式是错误的：

    ```
    let b = (let a = 8);
    ```

    rustc编译器会给出错误信息：

    ```
    a.rs:2:14: 2:17 error: expected identifier, found keyword `let`
    a.rs:2      let b = (let a = 8);
                         ^~~
    a.rs:2:18: 2:19 error: expected one of `!`, `)`, `,`, `.`, `::`, `{`, or an operator, found `a`
    a.rs:2      let b = (let a = 8);
                             ^
    ```

2. Item声明。是指函数（function）、结构体（structure）、类型别名（type）、静态变量（static）、特质（trait）、实现（implementation）或模块（module）的声明。这些声明可以嵌套在任意块（block）中。关于Item声明，Rust Reference中的描述如下：

> An item declaration statement has a syntactic form identical to an item declaration within a module. Declaring an item — a function, enumeration, structure, type, static, trait, implementation or module — locally within a statement block is simply a way of restricting its scope to a narrow region containing all of its uses; it is otherwise identical in meaning to declaring the item outside the statement block.

当然，这里不能展开讲这些Item都是如何声明的，详情请看RustPrimer的其他相关章节。

表达式语句

表达式语句，由一个表达式和一个分号组成，即在表达式后面加一个分号就将一个表达式转变为了一个语句。所以，有多少种表达式，就有多少种表达式语句。

rust有许多种表达式：

- 字面表达式（literal expression）

  ```
  ();          // unit type
  "hello";     // string type
  '1';         // character type
  15;           // integer type
  ```

- 元组表达式(Tuple expression)：

  ```
  (0.0, 4.5);
  ("a", 4usize, true);
  ```

  通常不使用一个元素的元组，不过如果你坚持的话，rust也是允许的，不过需要在元素后加一个逗号：

  ```
  (0,); // single-element tuple
  (0); // zero in parentheses
  ```

- 结构体表达式（structure expression） 由于结构体有多种形式，所以结构体表达式也有多种形式。

```
Point {x: 10.0, y: 20.0};
TuplePoint(10.0, 20.0);
let u = game::User {name: "Joe", age: 35, score: 100_000};
some_fn::<Cookie>(Cookie);
```

结构体表达式一般用于构造一个结构体对象，它除了以上从零构建的形式外，还可以在另一个对象的基础上进行构建：

```
let base = Point3d {x: 1, y: 2, z: 3};
Point3d {y: 0, z: 10, .. base};
```

- 块表达式（block expression）：块表达式就是用花括号 `{}` 括起来的一组表达式的集合，表达式间一般以分号分隔。块表达式的值，就是最后一个表达式的值。

```
let x: i32 = { println!("Hello."); 5 };
```

如果以语句结尾，则块表达式的值为 `()`：

```
let x: () = { println!("Hello."); };
```

- 范围表达式（range expression）：可以使用范围操作符 `..` 来构建范围对象（variant of `std::ops::Range`）：

```
1..2;   // std::ops::Range
3..;    // std::ops::RangeFrom
..4;    // std::ops::RangeTo
..;     // std::ops::RangeFull
```

- if表达式（if expression）：

```
let a = 9;
let b = if a%2 == 0 {"even"} else {"odd"};
```

- 除了以上这些外，还有许多，如：
    - path expression
    - mehond-call expression
    - field expression
    - array expression
    - index expression
    - unary operator expression
    - binary operator expression
    - return expression
    - grouped expression
    - match expression
    - if expression
    - lambda expression
    - ... ...

  这里无法详细展开，读者可以到Rust Reference去查看。

> 以上表达式语句中的部分例子引用自**Rust Reference**

高阶函数

高阶函数与普通函数的不同在于，它可以使用一个或多个函数作为参数，可以将函数作为返回值。rust的函数是first class type，所以支持高阶函数。而，由于rust是一个强类型的语言，如果要将函数作为参数或返回值，首先需要搞明白函数的类型。下面先说函数的类型，再说函数作为参数和返回值。

函数类型

前面说过，关键字 `fn` 可以用来定义函数。除此以外，它还用来构造函数类型。与函数定义主要的不同是，构造函数类型不需要函数名、参数名和函数体。在Rust Reference中的描述如下：

> The function type constructor fn forms new function types. A function type consists of a possibly-empty set of function-type modifiers (such as unsafe or extern), a sequence of input types and an output type.

来看一个简单例子：

```
fn inc(n: i32) -> i32 {//函数定义
  n + 1
}

type IncType = fn(i32) -> i32;//函数类型

fn main() {
  let func: IncType = inc;
  println!("3 + 1 = {}", func(3));
}
```

上例首先使用 `fn` 定义了 `inc` 函数，它有一个 `i32` 类型参数，返回 `i32` 类型的值。然后再用 `fn` 定义了一个函数类型，这个函数类型有i32类型的参数和i32类型的返回值，并用 `type` 关键字定义了它的别名 `IncType` 。在 `main` 函数中定义了一个变量 `func` ，其类型就为 `IncType` ，并赋值为 `inc` ，然后在 `pirntln` 宏中调用： `func(3)` 。可以看到， `inc` 函数的类型其实就是 `IncType` 。 这里有

一个问题，我们将 `inc` 赋值给了 `func` ，而不是 `&inc` ，这样是将 `inc` 函数的拥有权转给了 `func` 吗，赋值后还可以以 `inc()` 形式调用 `inc` 函数吗？先来看一个例子：

```
fn main() {
  let func: IncType = inc;
  println!("3 + 1 = {}", func(3));
  println!("3 + 1 = {}", inc(3));
}

type IncType = fn(i32) -> i32;

fn inc(n: i32) -> i32 {
  n + 1
}
```

我们将上例保存在rs源文件中，再用rustc编译，发现并没有报错，并且运行也得到我们想要的结果：

```
3 + 1 = 4
3 + 1 = 4
```

这说明，赋值时， `inc` 函数的所有权并没有被转移到 `func` 变量上，而是更像不可变引用。在rust中，函数的所有权是不能转移的，我们给函数类型的变量赋值时，赋给的一般是函数的指针，所以rust中的函数类型，就像是C/C++中的函数指针，当然，rust的函数类型更安全。可见，rust的函数类型，其实应该是属于指针类型（Pointer Type）。rust的Pointer Type有两种，一种为引用（Reference `&` ），另一种为原始指针（Raw pointer `*` ），详细内容请看Rust Reference 8.18 Pointer Types。而rust的函数类型应是引用类型，因为它是安全的，而原始指针则是不安全的，要使用原始指针，必须使用 `unsafe` 关键字声明。

函数作为参数

函数作为参数，其声明与普通参数一样。看下例：

```
fn main() {
  println!("3 + 1 = {}", process(3, inc));
  println!("3 - 1 = {}", process(3, dec));
}

fn inc(n: i32) -> i32 {
  n + 1
}

fn dec(n: i32) -> i32 {
  n - 1
}

fn process(n: i32, func: fn(i32) -> i32) -> i32 {
  func(n)
}
```

例子中， `process` 就是一个高阶函数，它有两个参数，一个类型为 `i32` 的 `n` ，另一个类型为 `fn(i32)->i32` 的函数 `func` ，返回一个 `i32` 类型的参数；它在函数体内以 `n` 作为参数调用 `func` 函数，返回 `func` 函数的返回值。运行可以得到以下结果：

```
3 + 1 = 4
3 - 1 = 2
```

不过，这不是函数作为参数的唯一声明方法，使用泛型函数配合特质（ `trait` ）也是可以的，因为rust的函数都会实现一个 `trait` : `FnOnce` 、 `Fn` 或 `FnMut` 。将上例中的 `process` 函数定义换成以下形式是等价的：

```
fn process<F>(n: i32, func: F) -> i32
    where F: Fn(i32) -> i32 {
    func(n)
}
```

函数作为返回值

函数作为返回值，其声明与普通函数的返回值类型声明一样。看例子：

```
fn main() {
   let a = [1,2,3,4,5,6,7];
   let mut b = Vec::<i32>::new();
   for i in &a {
       b.push(get_func(*i)(*i));
   }
   println!("{:?}", b);
}

fn get_func(n: i32) -> fn(i32) -> i32 {
    fn inc(n: i32) -> i32 {
        n + 1
    }
    fn dec(n: i32) -> i32 {
        n - 1
    }
    if n % 2 == 0 {
        inc
    } else {
        dec
    }
}
```

例子中的高阶函数为 `get_func` ，它接收一个i32类型的参数，返回一个类型为 `fn(i32) -> i32` 的函数，若传入的参数为偶数，返回 `inc` ，否则返回 `dec` 。这里需要注意的是， `inc` 函数和 `dec` 函数都定义在 `get_func` 内。在函数内定义函数在很多其他语言中是不支持的，不过rust支持，这也是rust灵活和强大的一个体现。不过，在函数中定义的函数，不能包含函数中（环境中）的变量，若要包含，应该闭包（详看13章 闭包）。 所以下例：

```
fn main() {
  let f = get_func();
  println!("{}", f(3));
}

fn get_func() -> fn(i32)->i32 {
  let a = 1;
  fn inc(n:i32) -> i32 {
    n + a
  }
  inc
}
```

使用rustc编译，会出现如下错误：

```
a.rs:9:9: 9:10 error: can't capture dynamic environment in a fn item; use the || { ... } closure form instead [E0434]
a.rs:9     n + a
           ^
a.rs:9:9: 9:10 error: unresolved name `a` [E0425]
a.rs:9     n + a
           ^
a.rs:9:9: 9:10 help: run `rustc --explain E0425` to see a detailed explanation
error: aborting due to 2 previous errors
```

模式匹配

除了我们常见的控制语句之外，Rust还提供了一个更加强大的关键字—— `match` 。但是，需要指出的一点是，match只是匹配，要发挥其全部威力，还需要模式的配合。本章，我们就将的对Rust的模式匹配进行一番探索。

本章内容：

match关键字

模式匹配，多出现在函数式编程语言之中，为其复杂的类型系统提供一个简单轻松的解构能力。比如从enum等数据结构中取出数据等等，但是在书写上，相对比较复杂。我们来看一个例子:

```
enum Direction {
    East,
    West,
    North,
    South,
}

fn main() {
    let dire = Direction::South;
    match dire {
        Direction::East => println!("East"),
        Direction::North | Direction::South => {
            println!("South or North");
        },
        _ => println!("West"),
    };
}
```

这是一个没什么实际意义的程序，但是能清楚的表达出match的用法。看到这里，你肯定能想起一个常见的控制语句—— `switch` 。没错，match可以起到和switch相同的作用。不过有几点需要注意：

1. match所罗列的匹配，必须穷举出其所有可能。当然，你也可以用 _ 这个符号来代表其余的所有可能性情况，就类似于switch中的 `default` 语句。
2. match的每一个分支都必须是一个表达式，并且，除非一个分支一定会触发panic，这些分支的所有表达式的最终返回值类型必须相同。

关于第二点，有的同学可能不明白。这么说吧，你可以把match整体视为一个表达式，既然是一个表达式，那么就一定能求得它的结果。因此，这个结果当然就可以被赋予一个变量咯。 看代码：

```
enum Direction {
    East,
    West,
    North,
    South,
}

fn main() {
    // let d_panic = Direction::South;
    let d_west = Direction::West;
    let d_str = match d_west {
        Direction::East => "East",
        Direction::North | Direction::South => {
            panic!("South or North");
        },
        _ => "West",
    };

    println!("{}", d_str);
}
```

解构初窥

match还有一个非常重要的作用就是对现有的数据结构进行解构，轻易的可以拿出其中的数据部分来。 比如，以下是比较常见的例子：

```
enum Action {
    Say(String),
    MoveTo(i32, i32),
    ChangeColorRGB(u16, u16, u16),
}

fn main() {
    let action = Action::Say("Hello Rust".to_string());
    match action {
        Action::Say(s) => {
            println!("{}", s);
        },
        Action::MoveTo(x, y) => {
            println!("point from (0, 0) move to ({}, {})", x, y)
;
        },
        Action::ChangeColorRGB(r, g, _) => {
            println!("change color into '(r:{}, g:{}, b:0)', 'b'
 has been ignored",
                r, g,
            );
        }
    }
}
```

有人说了，从这来看也并不觉得match有多神奇啊！别急，请看下一小节——>模式

模式

模式，是Rust另一个强大的特性。它可以被用在 `let` 和 `match` 表达式里面。相信大家应该还记得我们在复合类型中提到的关于在let表达式中解构元组的例子，实际上这就是一个模式。

```
let tup = (0u8, 1u8);
let (x, y) = tup;
```

而且我们需要知道的是，如果一个模式中出现了和当前作用域中已存在的同名的绑定，那么它会覆盖掉外部的绑定。比如：

```
let x = 1;
let c = 'c';

match c {
    x => println!("x: {} c: {}", x, c),
}

println!("x: {}", x);
```

它的输出结果是:

```
x: c c: c
x: 1
```

在以上代码中，match作用域里的 `x` 这个绑定被覆盖成了 `'c'` ，而出了这个作用域，绑定 `x` 又恢复为 `1` 。这和变量绑定的行为是一致的。

更强大的解构

在上一节里，我们初步了解了模式匹配在解构 `enum` 时候的便利性，事实上，在Rust中模式可以被用来对任何复合类型进行解构——struct/tuple/enum。现在我们要讲述一个复杂点的例子，对 `struct` 进行解构。

首先，我们可以对一个结构体进行标准的解构：

```
struct Point {
    x: i64,
    y: i64,
}
let point = Point { x: 0, y: 0 };
match point {
    Point { x, y } => println!("({},{})", x, y),
}
```

最终，我们拿到了 `Point` 内部的值。有人说了，那我想改个名字怎么办？很简单，你可以使用 `:` 来对一个struct的字段进行重命名，如下:

```
struct Point {
    x: i64,
    y: i64,
}
let point = Point { x: 0, y: 0 };
match point {
    Point { x: x1, y: y1} => println!("({},{})", x1, y1),
}
```

另外，有的时候我们其实只对某些字段感兴趣，就可以用 `..` 来省略其他字段。

```
struct Point {
    x: i64,
    y: i64,
}

let point = Point { x: 0, y: 0 };

match point {
    Point { y, .. } => println!("y is {}", y),
}
```

忽略和内存管理

总结一下，我们遇到了两种不同的模式忽略的情况—— `_` 和 `..` 。这里要注意，模式匹配中被忽略的字段是不会被 `move` 的，而且实现 `Copy` 的也会优先被Copy而不是被 `move` 。

说的有点拗口，上代码：

```
let tuple: (u32, String) = (5, String::from("five"));

let (x, s) = tuple;

// 以下行将导致编译错误，因为String类型并未实现Copy, 所以tuple被整体move掉了。
// println!("Tuple is: {:?}", tuple);

let tuple = (5, String::from("five"));

// 忽略String类型，而u32实现了Copy，则tuple不会被move
let (x, _) = tuple;

println!("Tuple is: {:?}", tuple);
```

范围和多重匹配

模式匹配可以被用来匹配单种可能，当然也就能被用来匹配多种情况：

范围

在模式匹配中，当我想要匹配一个数字(字符)范围的时候，我们可以用 `...` 来表示：

```
let x = 1;

match x {
    1 ... 10 => println!("一到十"),
    _ => println!("其它"),
}

let c = 'w';

match c {
    'a' ... 'z' => println!("小写字母"),
    'A' ... 'Z' => println!("大写字母"),
    _ => println!("其他字符"),
}
```

多重匹配

当我们只是单纯的想要匹配多种情况的时候，可以使用 `|` 来分隔多个匹配条件

```
let x = 1;

match x {
    1 | 2 => println!("一或二"),
    _ => println!("其他"),
}
```

ref 和 ref mut

前面我们了解到，当被模式匹配命中的时候，未实现 `Copy` 的类型会被默认的move掉，因此，原owner就不再持有其所有权。但是有些时候，我们只想要从中拿到一个变量的（可变）引用，而不想将其move出作用域，怎么做呢？答：用 `ref` 或者 `ref mut` 。

```
let mut x = 5;

match x {
    ref mut mr => println!("mut ref :{}", mr),
}
// 当然了……在let表达式里也能用
let ref mut mrx = x;
```

变量绑定

在模式匹配的过程内部，我们可以用 `@` 来绑定一个变量名，这在复杂的模式匹配中是再方便不过的，比如一个具名的范围匹配如下：

```
let x = 1u32;
match x {
    e @ 1 ... 5 | e @ 10 ... 15 => println!("get:{}", e),
    _ => (),
}
```

如代码所示，e绑定了x的值。

当然，变量绑定是一个极其有用的语法，下面是一个来自官方doc里的例子：

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}"
, a),
    _ => {}
}
```

输出：

```
Some("Steve")
```

后置条件

一个后置的if表达式可以被放在match的模式之后，被称为 `match guards` 。例如如下代码：

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("yes"),
    _ => println!("no"),
}
```

猜一下上面代码的输出？

答案是 `no` 。因为guard是后置条件，是整个匹配的后置条件：所以上面的式子表达的逻辑实际上是：

```
// 伪代码表示
IF y AND (x IN List[4, 5])
```

trait 和 trait对象

trait（特征）类似于其他语言中的**interface**或者**protocol**,指定一个实际类型必须满足的功能集合 与interface不同的地方在于，interface会隐藏具体实现类型，而trait不会。在rust中，隐藏实现类型可以由generic配合trait作出。

Rust中的trait：

- trait关键字
- trait对象

10.1 trait关键字

trait与具体类型

使用**trait**定义一个特征：

```
trait HasArea {
    fn area(&self) -> f64;
}
```

trait里面的函数可以没有函数体，实现代码交给具体实现它的类型去补充：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };
    println!("circle c has an area of {}", c.area());
}
```

注: **&self**表示的是**area**这个函数会将调用者的借代引用作为参数

这个程序会输出：

```
circle c has an area of 3.141592653589793
```

trait与泛型

> 我们了解了Rust中trait的定义和使用，接下来我们介绍一下它的使用场景，从中我们可以窥探出接口这特性带来的惊喜

我们知道泛型可以指任意类型，但有时这不是我们想要的，需要给它一些约束。

泛型的trait约束

```rust
use std::fmt::Debug;
fn foo<T: Debug>(s: T) {
    println!("{:?}", s);
}
```

`Debug` 是**Rust**内置的一个trait，为"{:?}"实现打印内容，函数 `foo` 接受一个泛型作为参数，并且约定其需要实现 `Debug`

多trait约束

可以使用多个trait对泛型进行约束：

```rust
use std::fmt::Debug;
fn foo<T: Debug + Clone>(s: T) {
    s.clone();
    println!("{:?}", s);
}
```

`<T: Debug + Clone>` 中 `Debug` 和 `Clone` 使用 `+` 连接，标示泛型 `T` 需要同时实现这两个trait。

where关键字

约束的trait增加后，代码看起来就变得诡异了，这时候需要使用 `where` 从句：

```
use std::fmt::Debug;
fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

// where 从句
fn foo<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

// 或者
fn foo<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

trait与内置类型

内置类型如：`i32`, `i64` 等也可以添加trait实现，为其定制一些功能：

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        *self as f64
    }
}

5.area();
```

这样的做法是有限制的。Rust 有一个“孤儿规则”：当你为某类型实现某 trait 的时候，必须要求类型或者 trait 至少有一个是在当前 crate 中定义的。你不能为第三方的类型实现第三方的 trait 。

在调用 trait 中定义的方法的时候，一定要记得让这个 trait 可被访问。

```
let mut f = std::fs::File::open("foo.txt").ok().expect("Couldn’t
 open foo.txt");
let buf = b"whatever"; //  buf: &[u8; 8]
let result = f.write(buf);
# result.unwrap();
```

这里是错误：

```
error: type `std::fs::File` does not implement any method in sco
pe named `write`
let result = f.write(buf);
               ^~~~~~~~~~
```

我们需要先use这个Write trait：

```
use std::io::Write;

let mut f = std::fs::File::open("foo.txt").expect("Couldn’t open
 foo.txt");
let buf = b"whatever";
let result = f.write(buf);
# result.unwrap(); // ignore the error
```

这样就能无错误地编译了。

trait的默认方法

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

`is_invalid` 是默认方法，`Foo` 的实现者并不要求实现它，如果选择实现它，会覆盖掉它的默认行为。

trait的继承

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

这样 `FooBar` 的实现者也要同时实现 `Foo` ：

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

derive属性

Rust提供了一个属性 `derive` 来自动实现一些trait，这样可以避免重复繁琐地实现他们，能被 `derive` 使用的trait包括： `Clone` , `Copy` , `Debug` , `Default` , `Eq` , `Hash` , `Ord` , `PartialEq` , `PartialOrd`

```
#[derive(Debug)]
struct Foo;

fn main() {
    println!("{:?}", Foo);
}
```

trait对象（trait object）

trait对象在**Rust**中是指使用指针封装了的 trait，比如 `&SomeTrait` 和 `Box<SomeTrait>`。

```
trait Foo { fn method(&self) -> String; }

impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } }
impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } }

fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
    let y = 8u8;
    do_something(&y);
}
```

`x: &Foo` 其中 `x` 是一个trait对象，这里用指针是因为 `x` 可以是任意实现 `Foo` 的类型实例，内存大小并不确定，但指针的大小是固定的。

trait对象的实现

`&SomeTrait` 类型和普通的指针类型 `&i32` 不同。它不仅包括指向真实对象的指针，还包括一个指向虚函数表的指针。它的内部实现定义在在 `std::raw` 模块中：

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

其中 `data` 是一个指向实际类型实例的指针， `vtable` 是一个指向实际类型对于该trait的实现的虚函数表：

`Foo` 的虚函数表类型：

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}
```

之前的代码可以解读为：

```
// u8:
// 这个函数只会被指向u8的指针调用
fn call_method_on_u8(x: *const ()) -> String {
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,

    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:
// 这个函数只会被指向String的指针调用
fn call_method_on_String(x: *const ()) -> String {
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}
```

```
static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};


let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // data存储实际值的引用
    data: &a,
    // vtable存储实际类型实现Foo的方法
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    data: &x,
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

对象安全

并不是所有的trait都能作为trait对象使用的，比如：

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

会有一个错误：

```
error: cannot convert to a trait object because trait `core::clo
ne::Clone` is not object-safe [E0038]
let o = &v as &Clone;
        ^~
note: the trait cannot require that `Self : Sized`
let o = &v as &Clone;
        ^~
```

让我来分析一下错误的原因：

```
pub trait Clone: Sized {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) { ... }
}
```

虽然 `Clone` 本身集成了 `Sized` 这个trait，但是它的方法 `fn clone(&self) -> Self` 和 `fn clone_from(&mut self, source: &Self) { ... }` 含有 `Self` 类型，而在使用trait对象方法的时候**Rust**是动态派发的，我们根本不知道这个trait对象的实际类型，它可以是任何一个实现了该trait的类型的值，所以 `Self` 在这里的大小不是 `Self: Sized` 的，这样的情况在**Rust**中被称为 `object-unsafe` 或者 `not object-safe` ，这样的trait是不能成为trait对象的。

总结：

如果一个 `trait` 方法是 `object safe` 的，它需要满足：

- 方法有 `Self: Sized` 约束， 或者
- 同时满足以下所有条件：
  - 没有泛型参数
  - 不是静态函数
  - 除了 `self` 之外的其它参数和返回值不能使用 `Self` 类型

如果一个 `trait` 是 `object-safe` 的，它需要满足：

- 所有的方法都是 `object-safe` 的，并且
- trait 不要求 `Self: Sized` 约束

参考stackoverflow object safe rfc

泛型

我们在编程中，通常有这样的需求，为多种类型的数据编写一个功能相同的函数，如两个数的加法，希望这个函数既支持i8、i16、 i32 ....float64等等，甚至自定义类型，在不支持泛型的编程语言中，我们通常要为每一种类型都编写一个函数，而且通常情况下函数名还必须不同，例如：

```
fn add_i8(a:i8, b:i8) -> i8 {
    a + b
}
fn add_i16(a:i16, b:i16) -> i16 {
    a + b
}
fn add_f64(a:f64, b:f64) -> f64 {
    a + b
}

// 各种其他add函数
// ...

fn main() {
    println!("add i8: {}", add_i8(2i8, 3i8));
    println!("add i16: {}", add_i16(20i16, 30i16));
    println!("add f64: {}", add_f64(1.23, 1.23));
}
```

如果有很多地方都需要支持多种类型，那么代码量就会非常大，而且代码也会非常臃肿，编程就真的变成了苦逼搬砖的工作，枯燥而乏味:D。 学过C++的人也许很容易理解泛型，但本教程面向的是Rust初学者，所以不会拿C++的泛型、多态和Rust进行对比，以免增加学习的复杂度和不必要的困扰，从而让Rust初学者更容易理解和接受Rust泛型。

概念

泛型程序设计是程序设计语言的一种风格或范式。允许程序员在强类型程序设计语言中编写代码时使用一些以后才指定的类型，在实例化时（instantiate）作为参数指明这些类型（在Rust中，有的时候类型还可以被编译器推导出来）。各种程序设计语言和其编译器、运行环境对泛型的支持均不一样。Ada, Delphi, Eiffel, Java, C#, F#, Swift, and Visual Basic .NET称之为泛型（generics）；ML, Scala and Haskell称之为参数多态（parametric polymorphism）；C++与D称之为模板。具有广泛影响的1994年版的《Design Patterns》一书称之为参数化类型（parameterized type）。

> 提示： 以上概念摘自《维基百科-泛型》

在编程的时候，我们经常利用多态。通俗的讲，多态就是好比坦克的炮管，既可以发射普通弹药，也可以发射制导炮弹（导弹），也可以发射贫铀穿甲弹，甚至发射子母弹，大家都不想为每一种炮弹都在坦克上分别安装一个专用炮管，即使生产商愿意，炮手也不愿意，累死人啊。所以在编程开发中，我们也需要这样“通用的炮管”，这个“通用的炮管”就是多态。

需要知道的是，泛型就是一种多态。

泛型主要目的是为程序员提供了编程的便利，减少代码的臃肿,同时极大丰富了语言本身的表达能力, 为程序员提供了一个合适的炮管。想想，一个函数，代替了几十个，甚至数百个函数，是一件多么让人兴奋的事情。 泛型，可以理解为具有某些功能共性的集合类型，如i8、i16、u8、f32等都可以支持add，甚至两个struct Point类型也可以add形成一个新的Point。

先让我们来看看标准库中常见的泛型Option，它的原型定义：

```
enum Option<T> {
    Some(T),
    None,
}
```

T就是泛型参数，这里的T可以换成A-Z任何你自己喜欢的字母。不过习惯上，我们用T表示Type，用E表示Error。T在具体使用的时候才会被实例化：

```
let a = Some(100.111f32);
```

编译器会自行推导出a为Option类型，也就是说Option中的T在这里是f32类型。

当然，你也可以显式声明a的类型，但必须保证和右值的类型一样，不然编译器会报"mismatched types"类型不匹配错误。

```
let a:Option<f32> = Some(100.111);  //编译自动推导右值中的100.111为f32类型。
let b:Option<f32> = Some(100.111f32);
let c:Option<f64> = Some(100.111);
let d:Option<f64> = Some(100.111f64);
```

泛型函数

至此，我们已经了解到泛型的定义和简单的使用了。 现在让我们用函数重写add操作：

```
use std::ops::Add;

fn add<T: Add<T, Output=T>>(a:T, b:T) -> T {
    a + b
}

fn main() {
    println!("{}", add(100i32, 1i32));
    println!("{}", add(100.11f32, 100.22f32));
}
```

> 输出: 101 200.33

`add<T: Add<T, Output=T>>(a:T, b:T) -> T` 就是我们泛型函数，返回值也是泛型T，Add<>中的含义可以暂时忽略，大体意思就是只要参数类型实现了Add trait，就可以被传入到我们的add函数，因为我们的add函数中有相加+操作，所以要求传进来的参数类型必须是可相加的，也就是必须实现了Add trait（具体参考std::ops::Add）。

自定义类型

上面的例子，add的都是语言内置的基础数据类型，当然我们也可以为自己自定义的数据结构类型实现add操作。

```
use std::ops::Add;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

// 为Point实现Add trait
impl Add for Point {
    type Output = Point; //执行返回值类型为Point
    fn add(self, p: Point) -> Point {
        Point{
            x: self.x + p.x,
            y: self.y + p.y,
        }
    }
}

fn add<T: Add<T, Output=T>>(a:T, b:T) -> T {
    a + b
}

fn main() {
    println!("{}", add(100i32, 1i32));
    println!("{}", add(100.11f32, 100.22f32));

    let p1 = Point{x: 1, y: 1};
    let p2 = Point{x: 2, y: 2};
    println!("{:?}", add(p1, p2));
}
```

> 输出: 101 200.33 Point { x: 3, y: 3 }

上面的例子稍微更复杂些了，只是我们增加了自定义的类型，然后让add函数依然可以在上面工作。如果对trait不熟悉，请查阅trait相关章节。

大家可能会疑问，那我们是否可以让Point也变成泛型的，这样Point的x和y也能够支持float类型或者其他类型，答案当然是可以的。

```
use std::ops::Add;

#[derive(Debug)]
struct Point<T: Add<T, Output = T>> { //限制类型T必须实现了Add trait，否则无法进行+操作。
    x: T,
    y: T,
}

impl<T: Add<T, Output = T>> Add for Point<T> {
    type Output = Point<T>;

    fn add(self, p: Point<T>) -> Point<T> {
        Point{
            x: self.x + p.x,
            y: self.y + p.y,
        }
    }
}

fn add<T: Add<T, Output=T>>(a:T, b:T) -> T {
    a + b
}

fn main() {
    let p1 = Point{x: 1.1f32, y: 1.1f32};
    let p2 = Point{x: 2.1f32, y: 2.1f32};
    println!("{:?}", add(p1, p2));

    let p3 = Point{x: 1i32, y: 1i32};
    let p4 = Point{x: 2i32, y: 2i32};
    println!("{:?}", add(p3, p4));
}
```

> 输出： Point { x: 3.2, y: 3.2 } Point { x: 3, y: 3 }

上面的列子更复杂了些，我们不仅让自定义的Point类型支持了add操作，同时我们也为Point做了泛型化。

当 `let p1 = Point{x: 1.1f32, y: 1.1f32};` 时，Point的T推导为f32类型，这样Point的x和y属性均成了f32类型。因为p1.x+p2.x，所以T类型必须支持Add trait。

总结

上面区区几十行的代码，却实现了非泛型语言百行甚至千行代码才能达到的效果，足见泛型的强大。

习题

1. Generic lines iterator

问题描述

有时候我们可能做些文本分析工作, 数据可能来源于外部或者程序内置的文本.

请实现一个 `parse` 函数, 只接收一个 lines iterator 为参数, 并输出每行.

要求既能输出内置的文本, 也能输出文件内容.

调用方式及输出参考

```
let lines = "some\nlong\ntext".lines()
parse(do_something_or_nothing(lines))
```

```
some
long
text
```

```
use std::fs:File;
use std::io::prelude::*;
use std::io::BufReader;
let lines = BufReader::new(File::open("/etc/hosts").unwrap()).li
nes()
parse(do_some_other_thing_or_nothing(lines))
```

```
127.0.0.1       localhost.localdomain   localhost
::1             localhost.localdomain   localhost
...
```

Hint

本书 `类型系统中的几个常见 trait` 章节中介绍的 AsRef, Borrow 等 trait 应该能派上用场.

所有权系统

概述

所有权系统（Ownership System）是Rust语言最基本最独特也是最重要的特性之一。

Rust追求的目标是内存安全与运行效率，但是它却没有golang, java, python等语言的内存垃圾回收机制GC。

Rust语言号称，只要编译通过就不会崩溃（内存安全）；拥有着零或者极小的运行时开销（运行效率）。这些优点也都得益于Rust的所有权系统。

所有权系统，包括三个重要的组成部分：

- **Ownership**（所有权）
- **Borrowing**（借用）
- **Lifetimes**（生命周期）

这三个特性之间相互关联，后面章节会依次全面讲解。

> 提示: Rust的所有权系统对很多初学者来说，可能会觉得难以理解，Rust的内存检查是在编译阶段完成，这个检查是非常严谨的，所以初学者在编译代码的时候，刚开始可能很难一次编译通过。
>
> 不过不要害怕：），当你一旦了解熟悉它后你会喜欢上它，并且在日后的编程中受益颇多。所有权系统需要读者慢慢体会其中的奥秘，学习过程中也可以参考官方文档。

所有权（Ownership）

在进入正题之前，大家先回忆下一般的编程语言知识。 对于一般的编程语言，通常会先声明一个变量，然后初始化它。 例如在C语言中：

```
int* foo() {
    int a;          // 变量a的作用域开始
    a = 100;
    char *c = "xyz";   // 变量c的作用域开始
    return &a;
}                   // 变量a和c的作用域结束
```

尽管可以编译通过，但这是一段非常糟糕的代码，现实中我相信大家都不会这么去写。变量a和c都是局部变量，函数结束后将局部变量a的地址返回，但局部变量 `a` 存在栈中，在离开作用域后，局部变量所申请的栈上内存都会被系统回收，从而造成了 `Dangling Pointer` 的问题。这是一个非常典型的内存安全问题。很多编程语言都存在类似这样的内存安全问题。再来看变量 `c` ， `c` 的值是常量字符串，存储于常量区，可能这个函数我们只调用了一次，我们可能不再想使用这个字符串，但 `xyz` 只有当整个程序结束后系统才能回收这片内存，这点让程序员是不是也很无奈？

> 备注：对于 `xyz` ，可根据实际情况，通过堆的方式，手动管理（申请和释放）内存。

所以，内存安全和内存管理通常是程序员眼中的两大头疼问题。令人兴奋的是，Rust却不再让你担心内存安全问题，也不用再操心内存管理的麻烦，那Rust是如何做到这一点的？请往下看。

绑定（Binding）

重要：首先必须强调下，准确地说Rust中并没有变量这一概念，而应该称为 `标识符` ，目标 `资源` (内存，存放value) `绑定` 到这个 `标识符` ：

```
{
    let x: i32;       // 标识符x, 没有绑定任何资源
    let y: i32 = 100; // 标识符y，绑定资源100
}
```

好了，我们继续看下以下一段Rust代码：

```
{
    let a: i32;
    println!("{}", a);
}
```

上面定义了一个i32类型的标识符 `a` ，如果你直接 `println!` ，你会收到一个error报错：

> error: use of possibly uninitialized variable: `a`

这是因为**Rust**并不会像其他语言一样可以为变量默认初始化值，**Rust**明确规定变量的初始值必须由程序员自己决定。

正确的做法：

```
{
    let a: i32;
    a = 100; //必须初始化a
    println!("{}", a);
}
```

其实， **`let`** 关键字并不只是声明变量的意思，它还有一层特殊且重要的概念-绑定。通俗的讲， `let` 关键字可以把一个标识符和一段内存区域做“绑定”，绑定后，这段内存就被这个标识符所拥有，这个标识符也成为这段内存的唯一所有者。 所以， `a = 100` 发生了这么几个动作，首先在栈内存上分配一个 `i32` 的资源，并填充值 `100` ，随后，把这个资源与 `a` 做绑定，让 `a` 成为资源的所有者(Owner)。

作用域

像C语言一样，Rust通过 `{}` 大括号定义作用域：

```
{
    {
        let a: i32 = 100;
    }
    println!("{}", a);
}
```

编译后会得到如下 `error` 错误：

> b.rs:3:20: 3:21 error: unresolved name `a` [E0425] b.rs:3 println!("{}", a);

像C语言一样，在局部变量离开作用域后，变量随即会被销毁；但不同是，**Rust**会连同变量绑定的内存，不管是否为常量字符串，连同所有者变量一起被销毁释放。所以上面的例子，a销毁后再次访问a就会提示无法找到变量 `a` 的错误。这些所有的一切都是在编译过程中完成的。

移动语义（move）

先看如下代码：

```
{
    let a: String = String::from("xyz");
    let b = a;
    println!("{}", a);
}
```

编译后会得到如下的报错：

> c.rs:4:20: 4:21 error: use of moved value: `a` [E0382] c.rs:4 println!("{}", a);

错误的意思是在 `println` 中访问了被 `moved` 的变量 `a` 。那为什么会有这种报错呢？具体含义是什么？ 在Rust中，和“绑定”概念相辅相成的另一个机制就是“转移move所有权”，意思是，可以把资源的所有权**(ownership)**从一个绑定转移（**move**）成另一个绑定，这个操作同样通过 `let` 关键字完成，和绑定不同的是， `=` 两边的左值和右值均为两个标识符：

```
语法：
    let 标识符A = 标识符B;  // 把“B”绑定资源的所有权转移给“A”
```

move前后的内存示意如下：

> **Before move:**
> a <=> 内存(地址：**A**，内容："xyz")
> **After move:**
> a
> b <=> 内存(地址：**A**，内容："xyz")

被move的变量不可以继续被使用。否则提示错误 `error: use of moved value` 。

这里有些人可能会疑问，move后，如果变量A和变量B离开作用域，所对应的内存会不会造成“Double Free”的问题？答案是否定的，**Rust**规定，只有资源的所有者销毁后才释放内存，而无论这个资源是否被多次 **`move`** ，同一时刻只有一个 **`owner`** ，所以该资源的内存也只会被 **`free`** 一次。 通过这个机制，就保证了内存安全。是不是觉得很强大？

Copy特性

有读者仿照“move”小节中的例子写了下面一个例子，然后说“a被move后是可以访问的”：

```
    let a: i32 = 100;
    let b = a;
    println!("{}", a);
```

编译确实可以通过，输出为 `100` 。这是为什么呢，是不是跟move小节里的结论相悖了？ 其实不然，这其实是根据变量类型是否实现 `Copy` 特性决定的。对于实现 `Copy` 特性的变量，在move时会拷贝资源到新内存区域，并把新内存区域的资源 `binding` 为 `b` 。

> **Before move:**
> a <=> 内存(地址：**A**，内容：100)
> **After move:**
> a <=> 内存(地址：**A**，内容：100)
> b <=> 内存(地址：**B**，内容：100)

move前后的 `a` 和 `b` 对应资源内存的地址不同。

在Rust中，基本数据类型(Primitive Types)均实现了Copy特性，包括i8, i16, i32, i64, usize, u8, u16, u32, u64, f32, f64, (), bool, char等等。其他支持Copy的数据类型可以参考官方文档的Copy章节。

浅拷贝与深拷贝

前面例子中move String和i32用法的差异，其实和很多面向对象编程语言中“浅拷贝”和“深拷贝”的区别类似。对于基本数据类型来说，“深拷贝”和“浅拷贝“产生的效果相同。对于引用对象类型来说，”浅拷贝“更像仅仅拷贝了对象的内存地址。 如果我们想实现对 `String` 的”深拷贝“怎么办？ 可以直接调用 `String` 的Clone特性实现对内存的值拷贝而不是简单的地址拷贝。

```
{
    let a: String = String::from("xyz");
    let b = a.clone();  // <-注意此处的clone
    println!("{}", a);
}
```

这个时候可以编译通过，并且成功打印"xyz"。

clone后的效果等同如下：

> **Before move:**
> a <=> 内存(地址：**A**，内容："xyz")
> **After move:**
> a <=> 内存(地址：**A**，内容："xyz")
> b <=> 内存(地址：**B**，内容："xyz")
> 注意，然后a和b对应的资源值相同，但是内存地址并不一样。

可变性

通过上面，我们已经已经了解了变量声明、值绑定、以及移动move语义等等相关知识，但是还没有进行过修改变量值这么简单的操作，在其他语言中看似简单到不值得一提的事却在Rust中暗藏玄机。 按照其他编程语言思维，修改一个变量的值：

```
let a: i32 = 100;
a = 200;
```

很抱歉，这么简单的操作依然还会报错：

> error: re-assignment of immutable variable `a` [E0384]
>
> :3 a = 200;

不能对不可变绑定赋值。如果要修改值，必须用关键字mut声明绑定为可变的：

```
let mut a: i32 = 100;  // 通过关键字mut声明a是可变的
a = 200;
```

想到“不可变”我们第一时间想到了 **`const`** 常量，但不可变绑定与 **`const`** 常量是完全不同的两种概念；首先，“不可变”准确地应该称为“不可变绑定”，是用来约束绑定行为的，“不可变绑定”后不能通过原“所有者”更改资源内容。

例如：

```
let a = vec![1, 2, 3];  //不可变绑定, a <=> 内存区域A(1,2,3)
let mut a = a;  //可变绑定， a <=> 内存区域A(1,2,3), 注意此a已非上句a
，只是名字一样而已
a.push(4);
println!("{:?}", a);  //打印：[1, 2, 3, 4]
```

“可变绑定”后，目标内存还是同一块，只不过，可以通过新绑定的a去修改这片内存了。

```
let mut a: &str = "abc";  //可变绑定, a <=> 内存区域A("abc")
a = "xyz";    //绑定到另一内存区域, a <=> 内存区域B("xyz")
println!("{:?}", a);  //打印："xyz"
```

上面这种情况不要混淆了， `a = "xyz"` 表示 `a` 绑定目标资源发生了变化。

其实，Rust中也有const常量，常量不存在“绑定”之说，和其他语言的常量含义相同：

```
const PI:f32 = 3.14;
```

可变性的目的就是严格区分绑定的可变性，以便编译器可以更好的优化，也提高了内存安全性。

高级Copy特性

在前面的小节有简单了解Copy特性，接下来我们来深入了解下这个特性。 Copy特性定义在标准库std::marker::Copy中：

```
pub trait Copy: Clone { }
```

一旦一种类型实现了Copy特性，这就意味着这种类型可以通过的简单的位(bits)拷贝实现拷贝。从前面知识我们知道“绑定”存在move语义（所有权转移），但是，一旦这种类型实现了Copy特性，会先拷贝内容到新内存区域，然后把新内存区域和这个标识符做绑定。

哪些情况下我们自定义的类型（如某个**Struct**等）可以实现**Copy**特性？ 只要这种类型的属性类型都实现了Copy特性，那么这个类型就可以实现Copy特性。 例如：

```
struct Foo {  //可实现Copy特性
    a: i32,
    b: bool,
}

struct Bar {  //不可实现Copy特性
    l: Vec<i32>,
}
```

因为 `Foo` 的属性 `a` 和 `b` 的类型 `i32` 和 `bool` 均实现了 `Copy` 特性，所以 `Foo` 也是可以实现Copy特性的。但对于 `Bar` 来说，它的属性 `l` 是 `Vec<T>` 类型，这种类型并没有实现 `Copy` 特性，所以 `Bar` 也是无法实

现 `Copy` 特性的。

那么我们如何来实现 **`Copy`** 特性呢？有两种方式可以实现。

1. 通过 **`derive`** 让**Rust**编译器自动实现

   ```
   #[derive(Copy, Clone)]
   struct Foo {
       a: i32,
       b: bool,
   }
   ```

   编译器会自动检查 `Foo` 的所有属性是否实现了 `Copy` 特性，一旦检查通过，便会为 `Foo` 自动实现 `Copy` 特性。

2. 手动实现 **`Clone`** 和 **`Copy`** **trait**

   ```
   #[derive(Debug)]
   struct Foo {
       a: i32,
       b: bool,
   }
   impl Copy for Foo {}
   impl Clone for Foo {
       fn clone(&self) -> Foo {
           Foo{a: self.a, b: self.b}
       }
   }
   fn main() {
       let x = Foo{ a: 100, b: true};
       let mut y = x;
       y.b = false;

       println!("{:?}", x);  //打印：Foo { a: 100, b: true }
       println!("{:?}", y);  //打印：Foo { a: 100, b: false }
   }
   ```

> 从结果我们发现 `let mut y = x` 后， `x` 并没有因为所有权 `move` 而出现不可访问错误。 因为 `Foo` 继承了 `Copy` 特性和 `Clone` 特性，所以例子中我们实现了这两个特性。

高级move

我们从前面的小节了解到， `let` 绑定会发生所有权转移的情况，
但 `ownership` 转移却因为资源类型是否实现 `Copy` 特性而行为不同：

```
let x: T = something;
let y = x;
```

- 类型 `T` 没有实现 `Copy` 特性： `x` 所有权转移到 `y` 。
- 类型 `T` 实现了 `Copy` 特性：拷贝 `x` 所绑定的 `资源` 为 `新资源` ，并把 `新资源` 的所有权绑定给 `y` ， `x` 依然拥有原资源的所有权。

move关键字

move关键字常用在闭包中，强制闭包获取所有权。

例子**1**：

```
fn main() {
    let x: i32 = 100;
    let some_closure = move |i: i32| i + x;
    let y = some_closure(2);
    println!("x={}, y={}", x, y);
}
```

> 结果： x=100, y=102

注意: 例子1是比较特别的，使不使用 move 对结果都没什么影响，因为 `x` 绑定的资源是 `i32` 类型，属于 `primitive type` ，实现了 `Copy trait` ，所以在闭包使用 `move` 的时候，是先 copy 了 `x` ，在 move 的时候是 move 了这份 clone 的 `x` ，所以后面的 `println!` 引用 `x` 的时候没有报错。

例子**2**：

```
fn main() {
    let mut x: String = String::from("abc");
    let mut some_closure = move |c: char| x.push(c);
    let y = some_closure('d');
    println!("x={:?}", x);
}
```

> 报错： error: use of moved value: `x` [E0382]
>
> :5 println!("x={:?}", x);

这是因为move关键字，会把闭包中的外部变量的所有权move到包体内，发生了所有权转移的问题，所以 `println` 访问 `x` 会如上错误。如果我们去掉 `println` 就可以编译通过。

那么，如果我们想在包体外依然访问x，即x不失去所有权，怎么办？

```
fn main() {
    let mut x: String = String::from("abc");
    {
        let mut some_closure = |c: char| x.push(c);
        some_closure('d');
    }
    println!("x={:?}", x);  //成功打印：x="abcd"
}
```

我们只是去掉了move，去掉move后，包体内就会对 `x` 进行了可变借用，而不是“剥夺” `x` 的所有权，细心的同学还注意到我们在前后还加了 `{}` 大括号作用域，是为了作用域结束后让可变借用失效，这样 `println` 才可以成功访问并打印我们期待的内容。

关于“**Borrowing**借用”知识我们会在下一个大节中详细讲解。

引用&借用（References&Borrowing）

如上所示，Owership让我们改变一个变量的值变得“复杂”，那能否像其他编程语言那样随意改变变量的值呢？答案是有的。

所有权系统允许我们通过“Borrowing”的方式达到这个目的。这个机制非常像其他编程语言中的“读写锁”，即同一时刻，只能拥有一个“写锁”，或只能拥有多个“读锁”，不允许“写锁”和“读锁”在同一时刻同时出现。当然这也是数据读写过程中保障一致性的典型做法。只不过Rust是在编译中完成这个(Borrowing)检查的，而不是在运行时，这也就是为什么其他语言程序在运行过程中，容易出现死锁或者野指针的问题。

通过**&**符号完成Borrowing：

```
fn main() {
    let x: Vec<i32> = vec!(1i32, 2, 3);
    let y = &x;
    println!("x={:?}, y={:?}", x, y);
}
```

Borrowing(**&x**)并不会发生所有权moved，所以println可以同时访问x和y。 通过引用，就可以对普通类型完成修改。

```
fn main() {
    let mut x: i32 = 100;
    {
        let y: &mut i32 = &mut x;
        *y += 2;
    }
    println!("{}", x);
}
```

借用与引用的区别

借用与引用是一种相辅相成的关系，若B是对A的引用，也可称之为B借用了A。

很相近对吧，但是借用一词本意为要归还。所以在Rust用引用时，一定要注意应该在何处何时正确的“归回”借用/引用。 最后面的“高级”小节会详细举例。

规则

1. 同一时刻，最多只有一个可变借用（&mut T），或者2。
2. 同一时刻，可有0个或多个不可变借用（&T）但不能有任何可变借用。
3. 借用在离开作用域后释放。
4. 在可变借用释放前不可访问源变量。

可变性

Borrowing也分“不可变借用”（默认，**&T**）和“可变借用”（**&mut T**）。

顾名思义，“不可变借用”是只读的，不可更新被引用的内容。

```
fn main() {
    let x: Vec<i32> = vec!(1i32, 2, 3);

    //可同时有多个不可变借用
    let y = &x;
    let z = &x;
    let m = &x;

    //ok
    println!("{:?}, {:?}, {:?}, {:?}", x, y, z, m);
}
```

再次强调下，同一时刻只能有一个可变借用(&mut T)，且被借用的变量本身必须有可变性：

```
fn main() {
    //源变量x可变性
    let mut x: Vec<i32> = vec!(1i32, 2, 3);

    //只能有一个可变借用
    let y = &mut x;
    // let z = &mut x; //错误
    y.push(100);

    //ok
    println!("{:?}", y);

    //错误，可变借用未释放，源变量不可访问
    // println!("{:?}", x);
}  //y在此处销毁
```

高级例子

下面的复杂例子，进行了详细的注释，即使看不懂也没关系，可以在完成Lifetimes（生命周期）的学习后再仔细思考本例子。

```
fn main() {
    let mut x: Vec<i32> = vec!(1i32, 2, 3);

    //更新数组
    //push中对数组进行了可变借用，并在push函数退出时销毁这个借用
    x.push(10);

    {
        //可变借用1
        let mut y = &mut x;
        y.push(100);

        //可变借用2，注意：此处是对y的借用，不可再对x进行借用，
        //因为y在此时依然存活。
        let z = &mut y;
        z.push(1000);

        println!("{:?}", z); //打印: [1, 2, 3, 10, 100, 1000]
    } //y和z在此处被销毁，并释放借用。


    //访问x正常
    println!("{:?}", x); //打印: [1, 2, 3, 10, 100, 1000]
}
```

总结

1. 借用不改变内存的所有者（Owner），借用只是对源内存的临时引用。
2. 在借用周期内，借用方可以读写这块内存，所有者被禁止读写内存；且所有者保证在有“借用”存在的情况下，不会释放或转移内存。
3. 失去所有权的变量不可以被借用（访问）。
4. 在租借期内，内存所有者保证不会释放/转移/可变租借这块内存，但如果是在非可变租借的情况下，所有者是允许继续非可变租借出去的。
5. 借用周期满后，所有者收回读写权限
6. 借用周期小于被借用者（所有者）的生命周期。

> 备注：　　借用周期，指的是借用的有效时间段。

生命周期（ Lifetime ）

下面是一个资源借用的例子：

```
fn main() {
    let a = 100_i32;

    {
        let x = &a;
    }  // x 作用域结束
    println!("{}", x);
}
```

编译时，我们会看到一个严重的错误提示：

> error: unresolved name `x`.

错误的意思是“无法解析 `x` 标识符”，也就是找不到 `x` , 这是因为像很多编程语言一样，Rust中也存在作用域概念，当资源离开离开作用域后，资源的内存就会被释放回收，当借用/引用离开作用域后也会被销毁，所以 `x` 在离开自己的作用域后，无法在作用域之外访问。

上面的涉及到几个概念：

- **Owner**: 资源的所有者 `a`
- **Borrower**: 资源的借用者 `x`
- **Scope**: 作用域，资源被借用/引用的有效期

强调下，无论是资源的所有者还是资源的借用/引用，都存在在一个有效的存活时间或区间，这个时间区间称为生命周期， 也可以直接以**Scope**作用域去理解。

所以上例子代码中的生命周期/作用域图示如下：

```
            {    a    {    x    }    *    }
所有者 a:         |________________________|
借用者 x:                   |____|            x = &a
  访问 x:                             |       失败：访问 x
```

可以看到，借用者 `x` 的生命周期是资源所有者 `a` 的生命周期的子集。但是 `x` 的生命周期在第一个 `}` 时结束并销毁，在接下来的 `println!` 中再次访问便会发生严重的错误。

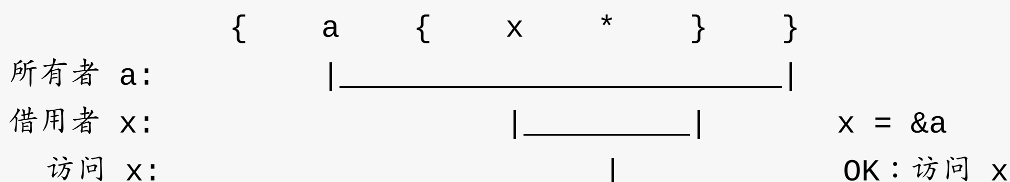
我们来修正上面的例子：

```
fn main() {
    let a = 100_i32;

    {
        let x = &a;
        println!("{}", x);
    }  // x 作用域结束

}
```

这里我们仅仅把 `println!` 放到了中间的 `{}`，这样就可以在 `x` 的生命周期内正常的访问 `x` ，此时的Lifetime图示如下：



```
             {    a    {    x    *    }    }
所有者 a:          |________________________|
借用者 x:                   |_________|        x = &a
  访问 x:                        |             OK：访问 x
```

隐式Lifetime

我们经常会遇到参数或者返回值为引用类型的函数：

```
fn foo(x: &str) -> &str {
    x
}
```

上面函数在实际应用中并没有太多用处，`foo` 函数仅仅接受一个 `&str` 类型的参数（`x` 为对某个 `string` 类型资源 `Something` 的借用），并返回对资源 `Something` 的一个新的借用。

实际上，上面函数包含该了隐性的生命周期命名，这是由编译器自动推导的，相当于：

```
fn foo<'a>(x: &'a str) -> &'a str {
    x
}
```

在这里，约束返回值的Lifetime必须大于或等于参数 `x` 的Lifetime。下面函数写法也是合法的：

```
fn foo<'a>(x: &'a str) -> &'a str {
    "hello, world!"
}
```

为什么呢？这是因为字符串"hello, world!"的类型是 `&'static str` ，我们知道 `static` 类型的Lifetime是整个程序的运行周期，所以她比任意传入的参数的Lifetime `'a` 都要长，即 `'static >= 'a` 满足。

在上例中Rust可以自动推导Lifetime，所以并不需要程序员显式指定Lifetime `'a` 。

`'a` 是什么呢？它是Lifetime的标识符，这里的 `a` 也可以用 `b` 、 `c` 、 `d` 、 `e` 、...，甚至可以用 `this_is_a_long_name` 等，当然实际编程中并不建议用这种冗长的标识符，这样会严重降低程序的可读性。 `foo` 后面的 `<'a>` 为Lifetime的声明，可以声明多个，如 `<'a, 'b>` 等等。

另外，除非编译器无法自动推导出Lifetime，否则不建议显式指定Lifetime标识符，会降低程序的可读性。

显式Lifetime

当输入参数为多个借用/引用时会发生什么呢？

```
fn foo(x: &str, y: &str) -> &str {
    if true {
        x
    } else {
        y
    }
}
```

这时候再编译，就没那么幸运了：

```
error: missing lifetime specifier [E0106]
fn foo(x: &str, y: &str) -> &str {
                            ^~~~
```

编译器告诉我们，需要我们显式指定Lifetime标识符，因为这个时候，编译器无法推导出返回值的Lifetime应该是比 `x` 长，还是比 `y` 长。虽然我们在函数中中用了 `if true` 确认一定可以返回 `x` ，但是要知道，编译器是在编译时候检查，而不是运行时，所以编译期间会同时检查所有的输入参数和返回值。

修复后的代码如下：

```
fn foo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

Lifetime推导

要推导Lifetime是否合法，先明确两点：

- 输出值（也称为返回值）依赖哪些输入值
- 输入值的Lifetime大于或等于输出值的Lifetime (准确来说：子集，而不是大于或等于)

Lifetime推导公式： 当输出值R依赖输入值X Y Z ...，当且仅当输出值的Lifetime为所有输入值的Lifetime交集的子集时，生命周期合法。

```
    Lifetime(R) ⊆ ( Lifetime(X) ∩ Lifetime(Y) ∩ Lifetime(Z) ∩ Li
fetime(...) )
```

对于例子1：

```
fn foo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

因为返回值同时依赖输入参数 `x` 和 `y` ，所以

```
    Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    'a ⊆ ('a ∩ 'a)  // 成立
```

定义多个Lifetime标识符

那我们继续看个更复杂的例子，定义多个Lifetime标识符：

```
fn foo<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

先看下编译，又报错了：

```
<anon>:5:3: 5:4 error: cannot infer an appropriate lifetime for
automatic coercion due to conflicting requirements [E0495]
<anon>:5         y
                 ^
<anon>:1:1: 7:2 help: consider using an explicit lifetime parame
ter as shown: fn foo<'a>(x: &'a str, y: &'a str) -> &'a str
<anon>:1 fn bar<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
<anon>:2     if true {
<anon>:3         x
<anon>:4     } else {
<anon>:5         y
<anon>:6     }
```

编译器说自己无法正确地推导返回值的Lifetime，读者可能会疑问，“我们不是已经指定返回值的Lifetime为 `'a` 了吗？"。

这儿我们同样可以通过生命周期推导公式推导：

因为返回值同时依赖 `x` 和 `y` ，所以

```
    Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    'a ⊆ ('a ∩ 'b)  //不成立
```

很显然，上面我们根本没法保证成立。

所以，这种情况下，我们可以显式地告诉编译器 `'b` 比 `'a` 长（ `'a` 是 `'b` 的子集），只需要在定义Lifetime的时候, 在 `'b` 的后面加上 `: 'a` , 意思是 `'b` 比 `'a` 长， `'a` 是 `'b` 的子集:

```
fn foo<'a, 'b: 'a>(x: &'a str, y: &'b str) -> &'a str {
    if true {
        x
    } else {
        y
    }
}
```

这里我们根据公式继续推导：

```
    条件：Lifetime(x) ⊆ Lifetime(y)
    推导：Lifetime(返回值) ⊆ ( Lifetime(x) ∩ Lifetime(y) )

    即：

    条件： 'a ⊆ 'b
    推导：'a ⊆ ('a ∩ 'b) // 成立
```

上面是成立的，所以可以编译通过。

推导总结

通过上面的学习相信大家可以很轻松完成Lifetime的推导，总之，记住两点：

1. 输出值依赖哪些输入值。
2. 推导公式。

Lifetime in struct

上面我们更多讨论了函数中Lifetime的应用，在 `struct` 中Lifetime同样重要。

我们来定义一个 `Person` 结构体：

```
struct Person {
    age: &u8,
}
```

编译时我们会得到一个error：

```
<anon>:2:8: 2:12 error: missing lifetime specifier [E0106]
<anon>:2     age: &str,
```

之所以会报错，这是因为Rust要确保 `Person` 的Lifetime不会比它的 `age` 借用长，不然会出现 `Dangling Pointer` 的严重内存问题。所以我们需要为 `age` 借用声明Lifetime：

```
struct Person<'a> {
    age: &'a u8,
}
```

不需要对 `Person` 后面的 `<'a>` 感到疑惑，这里的 `'a` 并不是指 `Person` 这个 `struct` 的Lifetime，仅仅是一个泛型参数而已， `struct` 可以有多个Lifetime参数用来约束不同的 `field` ，实际的Lifetime应该是所有 `field` Lifetime交集的子集。例如：

```
fn main() {
    let x = 20_u8;
    let stormgbs = Person {
                        age: &x,
                     };
}
```

这里，生命周期/Scope的示意图如下：

```
                    {    x     stormgbs       *      }
所有者 x:                |________________________|
所有者 stormgbs:                  |_______________|   'a
借用者 stormgbs.age:              |_______________|   stormgbs.age =
 &x
```

既然 `<'a>` 作为 `Person` 的泛型参数，所以在为 `Person` 实现方法时也需要加上 `<'a>` ，不然：

```
impl Person {
    fn print_age(&self) {
        println!("Person.age = {}", self.age);
    }
}
```

报错：

```
<anon>:5:6: 5:12 error: wrong number of lifetime parameters: exp
ected 1, found 0 [E0107]
<anon>:5 impl Person {
              ^~~~~~
```

正确的做法是：

```
impl<'a> Person<'a> {
    fn print_age(&self) {
        println!("Person.age = {}", self.age);
    }
}
```

这样加上 `<'a>` 后就可以了。读者可能会疑问，为什么 `print_age` 中不需要加上 `'a` ？这是个好问题。因为 `print_age` 的输出参数为 `()` ，也就是可以不依赖任何输入参数，所以编译器此时可以不必关心和推导Lifetime。即使是 `fn print_age(&self, other_age: &i32) {...}` 也可以编译通过。

如果 **`Person`** 的方法存在输出值（借用）呢？

```
impl<'a> Person<'a> {
    fn get_age(&self) -> &u8 {
        self.age
    }
}
```

`get_age` 方法的输出值依赖一个输入值 `&self` ，这种情况下，Rust编译器可以自动推导为：

```
impl<'a> Person<'a> {
    fn get_age(&'a self) -> &'a u8 {
        self.age
    }
}
```

如果输出值（借用）依赖了多个输入值呢？

```
impl<'a, 'b> Person<'a> {
    fn get_max_age(&'a self, p: &'a Person) -> &'a u8 {
        if self.age > p.age {
            self.age
        } else {
            p.age
        }
    }
}
```

类似之前的Lifetime推导章节，当返回值（借用）依赖多个输入值时，需显示声明Lifetime。和函数Lifetime同理。

其他

无论在函数还是在 `struct` 中，甚至在 `enum` 中，Lifetime理论知识都是一样的。希望大家可以慢慢体会和吸收，做到举一反三。

总结

Rust正是通过所有权、借用以及生命周期，以高效、安全的方式近乎完美地管理了内存。没有手动管理内存的负载和安全性，也没有GC造成的程序暂停问题。

闭包

闭包是什么？先来看看维基百科上的描述：

> 在计算机科学中，闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。
>
> 闭包的概念出现于60年代，最早实现闭包的程序语言是Scheme。之后，闭包被广泛使用于函数式编程语言如ML语言和LISP。很多命令式程序语言也开始支持闭包。

可以看到，第一句就已经说明了什么是闭包：闭包是引用了自由变量的函数。所以，闭包是一种特殊的函数。

在rust中，函数和闭包都是实现了 `Fn` 、 `FnMut` 或 `FnOnce` 特质（trait）的类型。任何实现了这三种特质其中一种的类型的对象，都是 可调用对象 ，都能像函数和闭包一样通过这样 `name()` 的形式调用， `()` 在rust中是一个操作符，操作符在rust中是可以重载的。rust的操作符重载是通过实现相应的 `trait` 来实现，而 `()` 操作符的相应 `trait` 就是 `Fn` 、 `FnMut` 和 `FnOnce` ，所以，任何实现了这三个 `trait` 中的一种的类型，其实就是重载了 `()` 操作符。关于 `Fn` 、 `FnMut` 和 `FnOnce` 的说明请看第二节闭包的实现。

本章主要分四节讲述：

闭包的语法

基本形式

闭包看起来像这样：

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

我们创建了一个绑定， `plus_one` ，并把它赋予一个闭包。闭包的参数位于管道（ `|` ）之中，而闭包体是一个表达式，在这个例子中， `x + 1` 。记住 `{}` 是一个表达式，所以我们也可以拥有包含多行的闭包：

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

你会注意到闭包的一些方面与用 `fn` 定义的常规函数有点不同。第一个是我们并不需要标明闭包接收和返回参数的类型。我们可以：

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

不过我们并不需要这么写。为什么呢？基本上，这是出于“人体工程学”的原因。因为为命名函数指定全部类型有助于像文档和类型推断，而闭包的类型则很少有文档因为它们是匿名的，并且并不会产生像推断一个命名函数的类型这样的“远距离错误”。

第二个的语法大同小异。我会增加空格来使它们看起来更像一点：

```
fn  plus_one_v1   (x: i32) -> i32 { x + 1 }
let plus_one_v2 = |x: i32| -> i32 { x + 1 };
let plus_one_v3 = |x: i32|          x + 1  ;
```

捕获变量

之所以把它称为“闭包”是因为它们“包含在环境中”（close over their environment）。这看起来像：

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

这个闭包， `plus_num` ，引用了它作用域中的 `let` 绑定： `num` 。更明确的说，它借用了绑定。如果我们做一些会与这个绑定冲突的事，我们会得到一个错误。比如这个：

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

错误是：

```
error: cannot borrow `num` as mutable because it is also borrowe
d as immutable
    let y = &mut num;
                 ^~~
note: previous borrow of `num` occurs here due to use in closure
; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` u
ntil the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

一个啰嗦但有用的错误信息！如它所说，我们不能取得一个 `num` 的可变借用因为闭包已经借用了它。如果我们让闭包离开作用域，我们可以：

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

如果你的闭包需要它，Rust会取得所有权并移动环境：

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

println!("{:?}", nums);
```

这会给我们：

```
note: `nums` moved into closure environment here because it has
type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-co
pyable
let takes_nums = || nums;
                    ^~~~~~~
```

`Vec<T>` 拥有它内容的所有权，而且由于这个原因，当我们在闭包中引用它时，我们必须取得 `nums` 的所有权。这与我们传递 `nums` 给一个取得它所有权的函数一样。

move闭包

我们可以使用 `move` 关键字强制使我们的闭包取得它环境的所有权：

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

现在，即便关键字是 `move` ，变量遵循正常的移动语义。在这个例子中， `5` 实现了 `Copy` ，所以 `owns_num` 取得一个 `5` 的拷贝的所有权。那么区别是什么呢？

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

那么在这个例子中，我们的闭包取得了一个 `num` 的可变引用，然后接着我们调用了 `add_num` ，它改变了其中的值，正如我们期望的。我们也需要将 `add_num` 声明为 `mut` ，因为我们会改变它的环境。

如果我们加上 `move` 修饰闭包，会发生些不同：

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

我们只会得到 `5` 。这次我们没有获取到外部的 `num` 的可变借用，我们实际上是把 `num` move 进了闭包。因为 `num` 具有 Copy 属性，因此发生 move 之后，以前的变量生命周期并未结束，还可以继续在 `assert_eq!` 中使用。我们打印的变量和闭包内的变量是独立的两个变量。如果我们捕获的环境变量不是 Copy 的，那么外部环境变量被 move 进闭包后， 它就不能继续在原先的函数中使用了，只能在闭包内使用。

不过在我们讨论获取或返回闭包之前，我们应该更多的了解一下闭包实现的方法。作为一个系统语言，Rust给予你了大量的控制你代码的能力，而闭包也是一样。

> 这部分引用自**The Rust Programming Language 中文版**

闭包的实现

Rust 的闭包实现与其它语言有些许不同。它们实际上是trait的语法糖。在这以前你会希望阅读trait章节，和trait对象。

都理解吗？很好。

理解闭包底层是如何工作的关键有点奇怪：使用 `()` 调用函数，像 `foo()` ，是一个可重载的运算符。到此，其它的一切都会明了。在Rust中，我们使用trait系统来重载运算符。调用函数也不例外。我们有三个trait来分别重载：

```
# mod foo {
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Outpu
t;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Sel
f::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::O
utput;
}
# }
```

你会注意到这些 trait 之间的些许区别，不过一个大的区别是 `self` ： `Fn` 获取 `&self` ， `FnMut` 获取 `&mut self` ，而 `FnOnce` 获取 `self` 。这包含了所有3种通过通常函数调用语法的 `self` 。不过我们将它们分在 3 个 trait 里，而不是单独的 1 个。这给了我们大量的对于我们可以使用哪种闭包的控制。

闭包的 `|| {}` 语法是上面 3 个 trait 的语法糖。Rust 将会为了环境创建一个结构体， `impl` 合适的 trait，并使用它。

> 这部分引用自**The Rust Programming Language 中文版**

闭包作为参数和返回值

闭包作为参数（Taking closures as arguments）

现在我们知道了闭包是 trait，我们已经知道了如何接受和返回闭包；就像任何其它的 trait！

这也意味着我们也可以选择静态或动态分发。首先，让我们写一个获取可调用结构的函数，调用它，然后返回结果：

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

我们传递我们的闭包，`|x| x + 2`，给 `call_with_one`。它正做了我们说的：它调用了闭包，`1` 作为参数。

让我们更深层的解析 `call_with_one` 的签名：

```
fn call_with_one<F>(some_closure: F) -> i32
#    where F : Fn(i32) -> i32 {
#    some_closure(1) }
```

我们获取一个参数，而它有类型 `F`。我们也返回一个 `i32`。这一部分并不有趣。下一部分是：

```
# fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {
#   some_closure(1) }
```

因为 `Fn` 是一个trait，我们可以用它限制我们的泛型。在这个例子中，我们的闭包取得一个 `i32` 作为参数并返回 `i32` ，所以我们用泛型限制是 `Fn(i32) -> i32` 。

还有一个关键点在于：因为我们用一个trait限制泛型，它会是单态的，并且因此，我们在闭包中使用静态分发。这是非常简单的。在很多语言中，闭包固定在堆上分配，所以总是进行动态分发。在Rust中，我们可以在栈上分配我们闭包的环境，并静态分发调用。这经常发生在迭代器和它们的适配器上，它们经常取得闭包作为参数。

当然，如果我们想要动态分发，我们也可以做到。trait对象处理这种情况，通常：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

现在我们取得一个trait对象，一个 `&Fn` 。并且当我们将我们的闭包传递给 `call_with_one` 时我们必须获取一个引用，所以我们使用 `&||` 。

函数指针和闭包

一个函数指针有点像一个没有环境的闭包。因此，你可以传递一个函数指针给任何函数除了作为闭包参数，下面的代码可以工作：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

fn add_one(i: i32) -> i32 {
    i + 1
}

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

在这个例子中，我们并不是严格的需要这个中间变量 `f` ，函数的名字就可以了：

```
let answer = call_with_one(&add_one);
```

返回闭包（Returning closures）

对于函数式风格代码来说在各种情况返回闭包是非常常见的。如果你尝试返回一个闭包，你可能会得到一个错误。在刚接触的时候，这看起来有点奇怪，不过我们会搞清楚。当你尝试从函数返回一个闭包的时候，你可能会写出类似这样的代码：

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

编译的时候会给出这一长串相关错误：

```
error: the trait `core::marker::Sized` is not implemented for th
e type
`core::ops::Fn(i32) -> i32` [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size
known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
error: the trait `core::marker::Sized` is not implemented for th
e type `core::ops::Fn(i32) -> i32` [E0277]
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size
known at compile-time
let f = factory();
    ^
```

为了从函数返回一些东西，Rust 需要知道返回类型的大小。不过 `Fn` 是一个 trait，它可以是各种大小(size)的任何东西。比如说，返回值可以是实现了 `Fn` 的任意类型。一个简单的解决方法是：返回一个引用。因为引用的大小(size)是固定的，因此返回值的大小就固定了。因此我们可以这样写：

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

不过这样会出现另外一个错误：

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~~
```

对。因为我们有一个引用，我们需要给它一个生命周期。不过我们的 `factory()` 函数不接收参数，所以省略不能用在这。我们可以使用什么生命周期呢？ `'static` ：

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

不过这样又会出现另一个错误：

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure@<anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~~
```

这个错误让我们知道我们并没有返回一个 `&'static Fn(i32) -> i32` ，而是返回了一个 `[closure <anon>:7:9: 7:20]` 。等等，什么？

因为每个闭包生成了它自己的环境 `struct` 并实现了 `Fn` 和其它一些东西，这些类型是匿名的。它们只在这个闭包中存在。所以Rust把它们显示为 `closure <anon>` ，而不是一些自动生成的名字。

这个错误也指出了返回值类型期望是一个引用，不过我们尝试返回的不是。更进一步，我们并不能直接给一个对象 `'static` 声明周期。所以我们换一个方法并通过 `Box` 装箱 `Fn` 来返回一个 trait 对象。这个几乎可以成功运行：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
# fn main() {
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
# }
```

这还有最后一个问题：

```
error: closure may outlive the current function, but it borrows
`num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~~
```

好吧，正如我们上面讨论的，闭包借用他们的环境。而且在这个例子中。我们的环境基于一个栈分配的 `5`，`num` 变量绑定。所以这个借用有这个栈帧的生命周期。所以如果我们返回了这个闭包，这个函数调用将会结束，栈帧也将消失，那么我们的闭包指向了被释放的内存环境！再有最后一个修改，我们就可以让它运行了：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
# fn main() {
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
# }
```

通过把内部闭包添加 `move` 关键字，我们强制闭包使用 move 的方式捕获环境变量。因为这里的 num 类型是 i32，实际上这里的 move 执行的是 copy, 这样一来，闭包就不再拥有指向环境的指针，而是完整拥有了被捕获的变量。并允许它离开我们的栈帧。

> 这部分引用自**The Rust Programming Language 中文版**

集合类型

就像C++的stl一样，Rust提供了一系列的基础且通用的容器类型。善用这些集合类型，可以让Rust编程更加方便轻松，但每种数据结构都会有其局限性，合理的选型方能维持更好的效率。

本章目录：

动态数组Vec

在第七章我们粗略介绍了一下Vec的用法。实际上，作为Rust中一个非常重要的数据类型，熟练掌握Vec的用法能大大提升我们在Rust世界中的编码能力。

特性及声明方式

和我们之前接触到的Array不同， `Vec` 具有动态的添加和删除元素的能力，并且能够以 `O(1)` 的效率进行随机访问。同时，对其尾部进行push或者pop操作的效率也是平摊 `O(1)` 的。 同时，有一个非常重要的特性（虽然我们编程的时候大部分都不会考量它）就是，Vec的所有内容项都是生成在堆空间上的，也就是说，你可以轻易的将Vec move出一个栈而不用担心内存拷贝影响执行效率——毕竟只是拷贝的栈上的指针。

另外的就是， `Vec<T>` 中的泛型 `T` 必须是 `Sized` 的，也就是说必须在编译的时候就知道存一个内容项需要多少内存。对于那些在编译时候未知大小的项（函数类型等），我们可以用 `Box` 将其包裹，当成一个指针。

new

我们可以用 `std::vec::Vec::new()` 的方式来声明一个Vec。

```
let mut v1: Vec<i32> = Vec::new();
```

这里需要注意的是， `new` 函数并没有提供一个能显式规定其泛型类型的参数，也就是说，上面的代码能根据 `v1` 的类型自动推导出 `Vec` 的泛型;但是，你不能写成如下的形式：

```
let mut v1 = Vec::new::<i32>();
// 与之对比的,collect函数就能指定：
// let mut v2 = (0i32..5).collect::<Vec<i32>>();
```

这是因为这两个函数的声明形式以及实现形式，在此，我们不做深究。

宏声明

相比调用new函数，Rust提供了一种更加直观便捷的方式声明一个动态数组： `vec!` 宏。

```
let v: Vec<i32> = vec![];

// 以下语句相当于：
// let mut temp = Vec::new();
// temp.push(1);
// temp.push(2);
// temp.push(3);
// let v = temp;
let v = vec![1, 2, 3];

let v = vec![0; 10]; //注意分号，这句话声明了一个 内容为10个0的动态数组
```

从迭代器生成

因为Vec实现了 `FromIterator` 这个trait，因此，借助collect，我们能将任意一个迭代器转换为Vec。

```
let v: Vec<_> = (1..5).collect();
```

访问及修改

随机访问

就像数组一样，因为Vec借助 `Index` 和 `IndexMut` 提供了随机访问的能力，我们通过 `[index]` 来对其进行访问，当然，既然存在随机访问就会出现越界的问题。而在Rust中，一旦越界的后果是极其严重的，可以导致Rust当前线程panic。因此，除非你确定自己在干什么或者在 `for` 循环中，不然我们不推荐通过下标访问。

以下是例子：

```
let a = vec![1, 2, 3];
assert_eq!(a[1usize], 2);
```

那么，Rust中有没有安全的下标访问机制呢？答案是当然有：—— `.get(n: usize)` （ `.get_mut(n: usize)` ） 函数。 对于一个数组，这个函数返回一个 `Option<&T>` (`Option<&mut T>`)，当Option==None的时候，即下标越界，其他情况下，我们能安全的获得一个Vec里面元素的引用。

```
let v =vec![1, 2, 3];
assert_eq!(v.get(1), Some(&2));
assert_eq!(v.get(3), None);
```

迭代器

对于一个可变数组，Rust提供了一种简单的遍历形式—— for 循环。 我们可以获得一个数组的引用、可变引用、所有权。

```
let v = vec![1, 2, 3];
for i in &v { .. } // 获得引用
for i in &mut v { .. } // 获得可变引用
for i in v { .. } // 获得所有权，注意此时Vec的属主将会被转移！！
```

但是，这么写很容易出现多层 `for` 循环嵌套，因此， `Vec` 提供了一个 `into_iter()` 方法，能显式地将自己转换成一个迭代器。然而迭代器怎么用呢？我们下一章将会详细说明。

push的效率研究

前面说到， `Vec` 有两个 `O(1)` 的方法，分别是 `pop` 和 `push` ，它们分别代表着将数据从尾部弹出或者装入。理论上来说，因为 `Vec` 是支持随机访问的，因此 `push` 效率应该是一致的。但是实际上，因为Vec的内部存在着内存拷贝和销毁，因此，如果你想要将一个数组，从零个元素开始，一个一个的填充直到最后生成一个非常巨大的数组的话，预先为其分配内存是一个非常好的办法。

这其中，有个关键的方法是reserve。

如下代码(注意：由于SystemTime API在1.8以后才稳定, 请使用1.8.0 stable 以及以上版本的rustc编译)：

```
use std::time;

fn push_1m(v: &mut Vec<usize>, total: usize) {
    let e = time::SystemTime::now();
    for i in 1..total {
        v.push(i);
    }
    let ed = time::SystemTime::now();
    println!("time spend: {:?}", ed.duration_since(e).unwrap());
}

fn main() {
    let mut v: Vec<usize> = vec![];
    push_1m(&mut v, 5_000_000);
    let mut v: Vec<usize> = vec![];
    v.reserve(5_000_000);
    push_1m(&mut v, 5_000_000);
}
```

在笔者自己的笔记本上，编译好了debug的版本，上面的代码跑出了：

```
➜  debug git:(master) ✗ time ./demo
time spend: Duration { secs: 0, nanos: 368875346 }
time spend: Duration { secs: 0, nanos: 259878787 }
./demo  0.62s user 0.01s system 99% cpu 0.632 total
```

好像并没有太大差异？然而切换到release版本的时候:

```
➜  release git:(master) ✗ time ./demo
time spend: Duration { secs: 0, nanos: 53389934 }
time spend: Duration { secs: 0, nanos: 24979520 }
./demo  0.06s user 0.02s system 97% cpu 0.082 total
```

注意消耗的时间的位数。可见，在去除掉debug版本的调试信息之后，是否预分配内存消耗时间降低了一倍！

这样的成绩，可见，预先分配内存确实有助于提升效率。

有人可能会问了，你这样纠结这点时间，最后不也是节省在纳秒级别的么，有意义么？当然有意义。

第一，纳秒也是时间，这还是因为这个测试的 `Vec` 只是最简单的内存结构。一旦涉及到大对象的拷贝，所花费的时间可就不一定这么少了。 第二，频繁的申请和删除堆空间，其内存一旦达到瓶颈的时候你的程序将会异常危险。

更多 `Vec` 的操作，请参照标准库的api。

哈希表 HashMap

和动态数组 `Vec` 一样，哈希表(HashMap)也是Rust内置的集合类型之一，同属 `std::collections` 模块下。

它提供了一个平均复杂度为 `O(1)` 的查询方法，是实现快速搜索必备的类型之一。

这里呢，主要给大家介绍一下HashMap的几种典型用法。

HashMap的要求

顾名思义, HashMap 要求一个可哈希（实现 Hash trait）的Key类型，和一个编译时知道大小的Value类型。 同时，Rust还要求你的Key类型必须是可比较的，在Rust中，你可以为你的类型轻易的加上编译器属性：

```
#[derive(PartialEq, Eq, Hash)]
```

这样，即可将你的类型转换成一个可以作为Hash的Key的类型。 但是，如果你想要自己实现 `Hash` 这个trait的话，你需要谨记两点：

- 1. 如果 Key1==Key2 ,那么一定有 Hash(Key1) == Hash(Key2)
- 1. 你的Hash函数本身不能改变你的Key值，否则将会引发一个逻辑错误（很难排查，遇到就完的那种）

什么？你看到 `std::hash::Hash` 这个 trait 中的函数没有 `&mut self` 的啊！但是，你不要忘了Rust中还有 `Cell` 和 `RefCell` 这种存在，他们提供了不可变对象的内部可变性，具体怎么变呢，请参照第20章。

另外，要保证你写的Hash函数不会被很轻易的碰撞，即 `Key1! = Key2` ，但 `Hash(Key1)==Hash(Key2)` ，碰撞的严重了，HashMap甚至有可能退化成链表！

这里笔者提议，别费劲，就按最简单的来就好。

增删改查

对于这种实用的类型，我们推荐用一个例子来解释：

```
use std::collections::HashMap;

// 声明
let mut come_from = HashMap::new();
// 插入
come_from.insert("WaySLOG", "HeBei");
come_from.insert("Marisa", "U.S.");
come_from.insert("Mike", "HuoGuo");

// 查找key
if !come_from.contains_key("elton") {
    println!("Oh, 我们查到了{}个人，但是可怜的Elton猫还是无家可归", com
e_from.len());
}

// 根据key删除元素
come_from.remove("Mike");
println!("Mike猫的家乡不是火锅！不是火锅！不是火锅！虽然好吃！");

// 利用get的返回判断元素是否存在
let who = ["MoGu", "Marisa"];
for person in &who {
    match come_from.get(person) {
        Some(location) => println!("{} 来自: {}", person, locatio
n),
        None => println!("{} 也无家可归啊.", person),
    }
}

// 遍历输出
println!("那么，所有人呢？");
for (name, location) in &come_from {
    println!("{}来自: {}", name, location);
}
```

这段代码输出：

```
Oh, 我们查到了3个人，但是可怜的Elton猫还是无家可归
Mike猫的家乡不是火锅！不是火锅！不是火锅！虽然好吃！
MoGu 也无家可归啊.
Marisa 来自: U.S.
那么，所有人呢？
Marisa来自: U.S.
WaySLOG来自: HeBei
```

entry

我们在编程的过程中，经常遇到这样的场景，统计一个字符串中所有的字符总共出现过几次。借助各种语言内置的Map类型我们总能完成这件事，但是完成的几乎都并不令人满意。很多人讨厌的一点是：为什么我要判断这个字符在字典中有没有出现，就要写一个大大的if条件！烦不烦？烦！于是，现代化的编程语言开始集成了类似Python里 `setdefault` 类似的特性（方法），下面是一段Python代码：

```python
val = {}
for c in "abcdefasdasdawe":
    val[c] = 1 + val.setdefault(c, 0)
print val
```

唔，总感觉怪怪的。那么Rust是怎么解决这个问题的呢？ 以下内容摘自标注库api注释：

```
use std::collections::HashMap;

let mut letters = HashMap::new();

for ch in "a short treatise on fungi".chars() {
    let counter = letters.entry(ch).or_insert(0);
    *counter += 1;
}

assert_eq!(letters[&'s'], 2);
assert_eq!(letters[&'t'], 3);
assert_eq!(letters[&'u'], 1);
assert_eq!(letters.get(&'y'), None);
```

Rust为我们提供了一个名叫 `entry` 的api，它很有意思，和Python相比，我们不需要在一次迭代的时候二次访问原map，只需要借用 entry 出来的Entry类型（这个类型持有原有HashMap的引用）即可对原数据进行修改。就语法来说，毫无疑问Rust在这个方面更加直观和具体。

迭代器

在Rust中，迭代器共分为三个部分：迭代器、适配器、消费者。

其中，迭代器本身提供了一个惰性的序列，适配器对这个序列进行诸如筛选、拼接、转换查找等操作，消费者则在前两者的基础上生成最后的数值集合。

但是，孤立的看这三者其实是没有意义的，因此，本章将在一个大节里联系写出三者。

迭代器

迭代器

从for循环讲起

我们在控制语句里学习了Rust的 `for` 循环表达式，我们知道，Rust的for循环实际上和C语言的循环语句是不同的。这是为什么呢？因为， `for` 循环不过是Rust编译器提供的语法糖！

首先，我们知道Rust有一个 `for` 循环能够依次对迭代器的任意元素进行访问，即：

```
for i in 1..10 {
    println!("{}", i);
}
```

这里我们知道， (1..10) 其本身是一个迭代器，我们能对这个迭代器调用 `.next()` 方法，因此， `for` 循环就能完整的遍历一个循环。 而对于 `Vec` 来说：

```
let values = vec![1,2,3];
for x in values {
    println!("{}", x);
}
```

在上面的代码中，我们并没有显式地将一个 `Vec` 转换成一个迭代器，那么它是如何工作的呢？现在就打开标准库翻api的同学可能发现了, `Vec` 本身并没有实现 `Iterator` ，也就是说，你无法对 `Vec` 本身调用 `.next()` 方法。但是，我们在搜索的时候，发现了 `Vec` 实现了 `IntoIterator` 的 trait。

其实， `for` 循环真正循环的，并不是一个迭代器(Iterator)，真正在这个语法糖里起作用的，是 `IntoIterator` 这个 trait。

因此，上面的代码可以被展开成如下的等效代码(只是示意，不保证编译成功):

```
let values = vec![1, 2, 3];

{
    let result = match IntoIterator::into_iter(values) {
        mut iter => loop {
            match iter.next() {
                Some(x) => { println!("{}", x); },
                None => break,
            }
        },
    };
    result
}
```

在这个代码里，我们首先对 `Vec` 调用 `into_iter` 来判断其是否能被转换成一个迭代器，如果能，则进行迭代。

那么，迭代器自己怎么办？

为此，Rust在标准库里提供了一个实现：

```
impl<I: Iterator> IntoIterator for I {
    // ...
}
```

也就是说，Rust为所有的迭代器默认的实现了 `IntoIterator` ，这个实现很简单，就是每次返回自己就好了。

也就是说：

任意一个 `Iterator` 都可以被用在 `for` 循环上！

无限迭代器

Rust支持通过省略高位的形式生成一个无限长度的自增序列，即：

```
let inf_seq = (1..).into_iter();
```

不过不用担心这个无限增长的序列撑爆你的内存，占用你的CPU，因为适配器的惰性的特性，它本身是安全的，除非你对这个序列进行 `collect` 或者 `fold` ！ 不过，我想聪明如你，不会犯这种错误吧！ 因此，想要应用这个，你需要用 `take` 或者 `take_while` 来截断他，必须？ 除非你将它当作一个生成器。当然了，那就是另外一个故事了。

消费者与适配器

说完了 `for` 循环，我们大致弄清楚了 `Iterator` 和 `IntoIterator` 之间的关系。下面我们来说一说消费者和适配器。

消费者是迭代器上一种特殊的操作，其主要作用就是将迭代器转换成其他类型的值，而非另一个迭代器。

而适配器，则是对迭代器进行遍历，并且其生成的结果是另一个迭代器，可以被链式调用直接调用下去。

由上面的推论我们可以得出: 迭代器其实也是一种适配器！

消费者

就像所有人都熟知的生产者消费者模型，迭代器负责生产，而消费者则负责将生产出来的东西最终做一个转化。一个典型的消费者就是 `collect` 。前面我们写过 `collect` 的相关操作，它负责将迭代器里面的所有数据取出，例如下面的操作：

```
let v = (1..20).collect(); //编译通不过的！
```

尝试运行上面的代码，却发现编译器并不让你通过。因为你没指定类型！指定什么类型呢？原来collect只知道将迭代器收集到一个实现了 `FromIterator` 的类型中去，但是，事实上实现这个 trait 的类型有很多（Vec, HashMap等），因此，collect没有一个上下文来判断应该将v按照什么样的方式收集！！

要解决这个问题，我们有两种解决办法：

1. 显式地标明 `v` 的类型:

```
let v: Vec<_> = (1..20).collect();
```

2. 显式地指定 `collect` 调用时的类型：

```
let v = (1..20).collect::<Vec<_>>();
```

当然，一个迭代器中还存在其他的消费者，比如取第几个值所用的 `.nth()` 函数，还有用来查找值的 `.find()` 函数，调用下一个值的 `next()` 函数等等，这里限于篇幅我们不能一一介绍。所以，下面我们只介绍另一个比较常用的消费者—— `fold` 。

当然了，提起Rust里的名字你可能没啥感觉，其实， `fold` 函数，正是大名鼎鼎的 MapReduce 中的 Reduce 函数(稍微有点区别就是这个Reduce是带初始值的)。

`fold` 函数的形式如下：

```
fold(base, |accumulator, element| .. )
```

我们可以写成如下例子：

```
let m = (1..20).fold(1u64, |mul, x| mul*x);
```

需要注意的是， `fold` 的输出结果的类型，最终是和 `base` 的类型是一致的（如果 `base` 的类型没指定，那么可以根据前面 `m` 的类型进行反推，除非 `m` 的类型也未指定），也就是说，一旦我们将上面代码中的 `base` 从 `1u64` 改成 `1` ，那么这行代码最终将会因为数据溢出而崩溃！

适配器

我们所熟知的生产消费的模型里，生产者所生产的东西不一定都会被消费者买账，因此，需要对原有的产品进行再组装。这个再组装的过程，就是适配器。因为适配器返回的是一个新的迭代器，所以可以直接用链式请求一直写下去。

前面提到了 Reduce 函数，那么自然不得不提一下另一个配套函数 —— `map` ：

熟悉Python语言的同学肯定知道，Python里内置了一个 `map` 函数，可以将一个迭代器的值进行变换，成为另一种。Rust中的 `map` 函数实际上也是起的同样的作用，甚至连调用方法也惊人的相似！

```
(1..20).map(|x| x+1);
```

上面的代码展示了一个“迭代器所有元素的自加一”操作，但是，如果你尝试编译这段代码，编译器会给你提示：

```
warning: unused result which must be used: iterator adaptors are
 lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on
 by default
(1..20).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

呀，这是啥？

因为，所有的适配器，都是惰性求值的！

也就是说，除非你调用一个消费者，不然，你的操作，永远也不会被调用到！

现在，我们知道了 `map` ，那么熟悉Python的人又说了，是不是还有 `filter` ！？答，有……用法类似， `filter` 接受一个闭包函数，返回一个布尔值，返回 `true` 的时候表示保留， `false` 丢弃。

```
let v: Vec<_> = (1..20).filter(|x| x%2 == 0).collect();
```

以上代码表示筛选出所有的偶数。

其他

上文中我们了解了迭代器、适配器、消费者的基本概念。下面将以例子来介绍Rust中的其他的适配器和消费者。

skip和take

`take(n)` 的作用是取前 `n` 个元素，而 `skip(n)` 正好相反，跳过前 `n` 个元素。

```
let v = vec![1, 2, 3, 4, 5, 6];
let v_take = v.iter()
    .cloned()
    .take(2)
    .collect::<Vec<_>>();
assert_eq!(v_take, vec![1, 2]);

let v_skip: Vec<_> = v.iter()
    .cloned()
    .skip(2)
    .collect();
assert_eq!(v_skip, vec![3, 4, 5, 6]);
```

zip 和 enumerate的恩怨情仇

`zip` 是一个适配器，他的作用就是将两个迭代器的内容压缩到一起，形成 `Iterator<Item=(ValueFromA, ValueFromB)>` 这样的新的迭代器；

```
let names = vec!["WaySLOG", "Mike", "Elton"];
let scores = vec![60, 80, 100];
let score_map: HashMap<_, _> = names.iter()
    .zip(scores.iter())
    .collect();
println!("{:?}", score_map);
```

而 `enumerate` ，熟悉的Python的同学又叫了：Python里也有！对的，作用也是一样的，就是把迭代器的下标显示出来，即：

```
let v = vec![1u64, 2, 3, 4, 5, 6];
let val = v.iter()
    .enumerate()
    // 迭代生成标，并且每两个元素剔除一个
    .filter(|&(idx, _)| idx % 2 == 0)
    // 将下标去除,如果调用unzip获得最后结果的话，可以调用下面这句，终止链
式调用
    // .unzip::<_,_, vec<_>, vec<_>>().1
    .map(|(idx, val)| val)
    // 累加 1+3+5 = 9
    .fold(0u64, |sum, acm| sum + acm);

println!("{}", val);
```

一系列查找函数

Rust的迭代器有一系列的查找函数，比如：

- `find()`：传入一个闭包函数，从开头到结尾依次查找能令这个闭包返回 `true` 的第一个元素，返回 `Option<Item>`
- `position()`：类似 `find` 函数，不过这次输出的是 `Option<usize>`，第几个元素。
- `all()`：传入一个函数，如果对于任意一个元素，调用这个函数返回 `false`，则整个表达式返回 `false`，否则返回 `true`
- `any()`：类似 `all()`，不过这次是任何一个返回 `true`，则整个表达式返回 `true`，否则 `false`
- `max()` 和 `min()`：查找整个迭代器里所有元素，返回最大或最小值的元素。注意：因为第七章讲过的 `PartialOrder` 的原因，`max` 和 `min` 作用在浮点数上会有不符合预期的结果。

以上，为常用的一些迭代器和适配器及其用法，仅作科普，对于这一章。我希望大家能够多练习去理解，而不是死记硬背。

好吧，留个习题：

习题

利用迭代器生成一个升序的长度为10的水仙花数序列，然后对这个序列进行逆序,并输出

模块和包系统、Prelude

前言

随着工程的增大，把所有代码写在一个文件里面，是一件极其初等及愚蠢的作法。大体来讲，它有如下几个缺点：

1. 文件大了，编辑器打开慢；
2. 所有代码放在同一个文件中，无法很好地利用现代多窗口编辑器，同时查看编辑相关联的两个代码片断；
3. 代码数量过多，查找某一个关键词过慢，定位到某一行代码的效率会大大降低；
4. 会大大增加上翻下翻的频率，导致你的鼠标中间滚轮易坏；
5. 不断地上翻下翻，会导致你头晕；
6. 头晕了，就容易写出错误的代码，甚至改错文件中的某一行（相似的地方，改错地方了）；
7. 出现bug，根据错误反馈，知道是哪一片逻辑的问题，但不容易快速定位；

因此，模块是几乎所有语言的基础设施，尽管叫法各有不同。

包和模块

包（crate）

Rust 中， `crate` 是一个独立的可编译单元。具体说来，就是一个或一批文件（如果是一批文件，那么有一个文件是这个 crate 的入口）。它编译后，会对应着生成一个可执行文件或一个库。

执行 `cargo new foo` ，会得到如下目录层级：

```
foo
├── Cargo.toml
└── src
    └── lib.rs
```

这里， `lib.rs` 就是一个 crate（入口），它编译后是一个库。一个工程下可以包含不止一个 crate，本工程只有一个。

执行 `cargo new --bin bar` ，会得到如下目录层级：

```
bar
├── Cargo.toml
└── src
    └── main.rs
```

这里， `main.rs` 就是一个 crate（入口），它编译后是一个可执行文件。

模块（module）

Rust 提供了一个关键字 `mod` ，它可以在一个文件中定义一个模块，或者引用另外一个文件中的模块。

关于模块的一些要点：

1. 每个 crate 中，默认实现了一个隐式的 `根模块（root module）` ；

2. 模块的命名风格也是 `lower_snake_case`，跟其它的 Rust 的标识符一样；
3. 模块可以嵌套；
4. 模块中可以写任何合法的 Rust 代码；

在文件中定义一个模块

比如，在上述 `lib.rs` 中，我们写上如下代码：

```
mod aaa {
    const X: i32 = 10;

    fn print_aaa() {
        println!("{}", 42);
    }
}
```

我们可以继续写如下代码：

```
mod aaa {
    const X: i32 = 10;

    fn print_aaa() {
        println!("{}", 42);
    }

    mod BBB {
        fn print_bbb() {
            println!("{}", 37);
        }
    }
}
```

还可以继续写：

```
mod aaa {
    const X: i32 = 10;

    fn print_aaa() {
        println!("{}", 42);
    }

    mod bbb {
        fn print_bbb() {
            println!("{}", 37);
        }
    }
}

mod ccc {
    fn print_ccc() {
        println!("{}", 25);
    }

}
```

模块的可见性

我们前面写了一些模块，但实际上，我们写那些模块，目前是没有什么作用的。写模块的目的一是为了分隔逻辑块，二是为了提供适当的函数，或对象，供外部访问。而模块中的内容，默认是私有的，只有模块内部能访问。

为了让外部能使用模块中 item，需要使用 `pub` 关键字。外部引用的时候，使用 `use` 关键字。例如：

```
mod ccc {
    pub fn print_ccc() {
        println!("{}", 25);
    }
}

fn main() {
    use ccc::print_ccc;

    print_ccc();
    // 或者
    ccc::print_ccc();
}
```

规则很简单，一个 item（函数，绑定，Trait 等），前面加了 `pub` ，那么就它变成对外可见（访问，调用）的了。

引用外部文件模块

通常，我们会在单独的文件中写模块内容，然后使用 `mod` 关键字来加载那个文件作为我们的模块。

比如，我们在 `src` 下新建了文件 `aaa.rs` 。现在目录结构是下面这样子：

```
foo
├── Cargo.toml
└── src
    └── aaa.rs
    └── main.rs
```

我们在 `aaa.rs` 中，写上：

```
pub fn print_aaa() {
    println!("{}", 25);
}
```

在 `main.rs` 中，写上：

```
mod aaa;

use aaa::print_aaa;

fn main () {
    print_aaa();
}
```

编译后，生成一个可执行文件。

细心的朋友会发现，`aaa.rs` 中，没有使用 `mod xxx {}` 这样包裹起来，是因为 `mod xxx;` 相当于把 `xxx.rs` 文件用 `mod xxx {}` 包裹起来了。初学者往往会多加一层，请注意。

多文件模块的层级关系

Rust 的模块支持层级结构，但这种层级结构本身与文件系统目录的层级结构是解耦的。

`mod xxx;` 这个 `xxx` 不能包含 `::` 号。也即在这个表达形式中，是没法引用多层结构下的模块的。也即，你不可能直接使用 `mod a::b::c::d;` 的形式来引用 `a/b/c/d.rs` 这个模块。

那么，Rust 的多层模块遵循如下两条规则：

1. 优先查找`xxx.rs` 文件
    i. `main.rs` 、 `lib.rs` 、 `mod.rs` 中的`mod xxx;` 默认优先查找同级目录下的 `xxx.rs` 文件；
    ii. 其他文件`yyy.rs` 中的`mod xxx;`默认优先查找同级目录的`yyy` 目录下的 `xxx.rs` 文件；
2. 如果 `xxx.rs` 不存在，则查找 `xxx/mod.rs` 文件，即 `xxx` 目录下的 `mod.rs` 文件。

上述两种情况，加载成模块后，效果是相同的。Rust 就凭这两条规则，通过迭代使用，结合 `pub` 关键字，实现了对深层目录下模块的加载；

下面举个例子，现在我们建了一个测试工程，目录结构如下：

```
src
├── a
|   ├── b
|   |   ├── c
|   |   |   ├── d.rs
|   |   |   └── mod.rs
|   |   └── mod.rs
|   └── mod.rs
└── main.rs
```

`a/b/c/d.rs` 文件内容：

```
pub fn print_ddd() {
    println!("i am ddd.");
}
```

`a/b/c/mod.rs` 文件内容：

```
pub mod d;
```

`a/b/mod.rs` 文件内容：

```
pub mod c;
```

`a/mod.rs` 文件内容：

```
pub mod b;
```

`main.rs` 文件内容：

```
mod a;

use a::b::c::d;

fn main() {
    d::print_ddd();
}
```

输出结果为： `i am ddd.`

仔细理解本例子，就明白 Rust 的层级结构模块的用法了。

至于为何 Rust 要这样设计，有几下几个原因：

1. Rust 本身模块的设计是与操作系统文件系统目录解耦的，因为 Rust 本身可用于操作系统的开发；
2. Rust 中的一个文件内，可包含多个模块，直接将 `a::b::c::d` 映射到 `a/b/c/d.rs` 会引起一些歧义；
3. Rust 一切从安全性、显式化立场出发，要求引用路径中的每一个节点，都是一个有效的模块，比如上例， `d` 是一个有效的模块的话，那么，要求 `c, b, a` 分别都是有效的模块，可单独引用。

路径

前面我们提到，一个 crate 是一个独立的可编译单元。它有一个入口文件，这个入口文件是这个 crate（里面可能包含若干个 module）的模块根路径。整个模块的引用，形成一个链，每个模块，都可以用一个精确的路径（比如： `a::b::c::d` ）来表示；

与文件系统概念类似，模块路径也有相对路径和绝对路径的概念。为此，Rust 提供了 `self` 和 `super` 两个关键字。

`self` 在路径中，有两种意思：

1. `use self::xxx` 表示，加载当前模块中的 `xxx` 。此时 self 可省略；
2. `use xxx::{self, yyy}` ，表示，加载当前路径下模块 `xxx` 本身，以及模块 `xxx` 下的 `yyy` ；

`super` 表示，当前模块路径的上一级路径，可以理解成父模块。

```
use super::xxx;
```

表示引用父模块中的 `xxx` 。

另外，还有一种特殊的路径形式：

```
::xxx::yyy
```

它表示，引用根路径下的 `xxx::yyy` ，这个根路径，指的是当前 crate 的根路径。

路径中的 `*` 符号：

```
use xxx::*;
```

表示导入 `xxx` 模块下的所有可见 item（加了 pub 标识的 item）。

Re-exporting

我们可以结合使用 `pub use` 来实现 `Re-exporting` 。 `Re-exporting` 的字面意思就是 `重新导出` 。它的意思是这样的，把深层的 item 导出到上层目录中，使调用的时候，更方便。接口设计中会大量用到这个技术。

还是举上面那个 `a::b::c::d` 的例子。我们在 `main.rs` 中，要调用 `d` ，得使用 `use a::b::c::d;` 来调用。而如果我们修改 `a/mod.rs` 文件为：

`a/mod.rs` 文件内容：

```
pub mod b;
pub use b::c::d;
```

那么，我们在 `main.rs` 中，就可以使用 `use a::d;` 来调用了。从这个例子来看没觉得方便多少。但是如果开发的一个库中有大量的内容，而且是在不同层次的模块中。那么，通过统一导出到一个地方，就能大大方便接口使用者。

加载外部 crate

前面我们讲的，都是在当前 crate 中的技术。真正我们在开发时，会大量用到外部库。外部库是通过

```
extern crate xxx;
```

这样来引入的。

注：要使上述引用生效，还必须在 `Cargo.toml` 的 `dependecies` 段，加上 `xxx="version num"` 这种依赖说明，详情见 `Cargo 项目管理` 这一章。

引入后，就相当于引入了一个符号 `xxx`，后面可以直接以这个 `xxx` 为根引用这个 crate 中的 item：

```
extern crate xxx;

use xxx::yyy::zzz;
```

引入的时候，可以通过 `as` 关键字重命名。

```
extern crate xxx as foo;

use foo::yyy::zzz;
```

Prelude

Rust 的标准库，有一个 `prelude` 子模块，这里面包含了默认导入（std 库是默认导入的，然后 std 库中的 prelude 下面的东西也是默认导入的）的所有符号。

大体上有下面一些内容：

```
std::marker::{Copy, Send, Sized, Sync}
std::ops::{Drop, Fn, FnMut, FnOnce}
std::mem::drop
std::boxed::Box
std::borrow::ToOwned
std::clone::Clone
std::cmp::{PartialEq, PartialOrd, Eq, Ord}
std::convert::{AsRef, AsMut, Into, From}
std::default::Default
std::iter::{Iterator, Extend, IntoIterator, DoubleEndedIterator, ExactSizeIterator}
std::option::Option::{self, Some, None}
std::result::Result::{self, Ok, Err}
std::slice::SliceConcatExt
std::string::{String, ToString}
std::vec::Vec
```

pub restricted

概览

这是 rust1.18 新增的一个语法。在此之前的版本，`item` 只有 `pub` /non- `pub` 两种分类，pub restricted 这个语法用来扩展 `pub` 的使用，使其能够指定想要的作用域(可见范围)，详情参见RFC 1422-pub-restricted.md。

在 Rust 中 `crate` 是一个模块树，可以通过表达式 `pub(crate) item;` 来限制 `item` 只在当前 `crate` 中可用，在当前 `crate` 的其他子树中，可以通过 `use + path` 的语法来引用 `item`。

设计动因

Rust1.18 之前，如果我们想要设计一个 item `x` 可以在多处使用，那么有两种方法：

- 在根目录中定义一个非 `pub` item；
- 在子模块中定义一个 `pub` item，同时通过 `use` 将这个项目引用到根目录。

但是，有时候这两种方法都并不是我们想要的。在一些情况下，我们希望对于某些特定的模块，该项目可见，而其他模块不允许使用。

下面我们来看一个例子：

```
// Intent: `a` exports `I`, `bar`, and `foo`, but nothing else.
pub mod a {
    pub const I: i32 = 3;

    // `semisecret` will be used "many" places within `a`, but
    // is not meant to be exposed outside of `a`.
    fn semisecret(x: i32) -> i32  { use self::b::c::J; x + J }

    pub fn bar(z: i32) -> i32 { semisecret(I) * z }
    pub fn foo(y: i32) -> i32 { semisecret(I) + y }

    mod b {
        mod c {
            const J: i32 = 4; // J is meant to be hidden from th
e outside world.
        }
    }
}
```

这段代码编译无法通过，因为 `J` 无法在 `mod c` 的外部访问，而 `fn semisecret` 尝试在 `mod a` 中访问 `J` .

在 rust1.18 之前，保持 `J` 私有，并能够让 `a` 使用 `fn semisecret` 的正确写法是，将 `fn semisecret` 移动到 `mod c` 中，并将其 `pub` ，之后根据需要可以重新导出 `semisecret` 。(如果不需要保持 `J` 的私有化，那么可以对其进行 `pub` ，之后可以在 `b` 中 `pub use self::c::J` 或者直接 `pub c`)

```
// Intent: `a` exports `I`, `bar`, and `foo`, but nothing else.
pub mod a {
    pub const I: i32 = 3;

    // `semisecret` will be used "many" places within `a`, but
    // is not meant to be exposed outside of `a`.
    // (If we put `pub use` here, then *anyone* could access it.)

    use self::b::semisecret;

    pub fn bar(z: i32) -> i32 { semisecret(I) * z }
    pub fn foo(y: i32) -> i32 { semisecret(I) + y }

    mod b {
        pub use self::c::semisecret;
        mod c {
            const J: i32 = 4; // J is meant to be hidden from th
e outside world.
            pub fn semisecret(x: i32) -> i32  { x + J }
        }
    }
}
```

这种情况可以正常工作，但是，这里有个严重的问题：没有人能够十分清晰的说明 `pub fn semiseret` 使用到了哪些地方，需要通过上下文进行判断：

1. 所有可以访问 `semiseret` 的模块；
2. 在所有可以访问 `semiseret` 的模块中，是否存在 `semiseret` 的 re-export;

同时，如果在 `a` 中使用 `pub use self::b::semisecret` ，那么所有人都可以通过 `use` 访问 `fn semiseret`，但是实际上，这个函数只需要让 `mod a` 访问就可以了。

pub restricted 的使用

Syntax

old:

```
VISIBILITY ::= <empty> | `pub`
```

new:

```
VISIBILITY ::= <empty> | `pub` | `pub` `(` USE_PATH `)` | `pub`
`(` `crate` `)`
```

pub(restriction) 意味着对 item，method，field等的定义加以可见范围（作用域）的限制。

可见范围（作用域）分为所有 crate (无限制)，当前 crate，当前 crate 中的子模块的绝对路径。被限制的东西不能在其限制范围之外直接使用。

- `pub` 无明确指定意味着无限制；
- `pub(crate)` 当前 crate 有效；
- `pub(in <path>)` 在 `<path>` 表示的模块中有效。

修改示例

```
// Intent: `a` exports `I`, `bar`, and `foo`, but nothing else.
pub mod a {
    pub const I: i32 = 3;

    // `semisecret` will be used "many" places within `a`, but
    // is not meant to be exposed outside of `a`.
    // (`pub use` would be *rejected*; see Note 1 below)
    use self::b::semisecret;

    pub fn bar(z: i32) -> i32 { semisecret(I) * z }
    pub fn foo(y: i32) -> i32 { semisecret(I) + y }

    mod b {
        pub(in a) use self::c::semisecret;
        mod c {
            const J: i32 = 4; // J is meant to be hidden from th
e outside world.

            // `pub(in a)` means "usable within hierarchy of `mo
d a`, but not
            // elsewhere."
            pub(in a) fn semisecret(x: i32) -> i32  { x + J }
        }
    }
}
```

Note 1: 如果改成下面这种方式，编译器会报错:

```
pub mod a { [...] pub use self::b::semisecret; [...] }
```

因为 `pub(in a) fn semisecret` 说明这个函数只能在 `a` 中使用，不允许 `pub` 出 `a` 的范围。

限制字段示例

```rust
mod a {
    #[derive(Default)]
    struct Priv(i32);

    pub mod b {
        use a::Priv as Priv_a;

        #[derive(Default)]
        pub struct F {
            pub    x: i32,
                   y: Priv_a,
            pub(in a) z: Priv_a,
        }

        #[derive(Default)]
        pub struct G(pub i32, Priv_a, pub(in a) Priv_a);

        // ... accesses to F.{x,y,z} ...
        // ... accesses to G.{0,1,2} ...
    }
    // ... accesses to F.{x,z} ...
    // ... accesses to G.{0,2} ...
}

mod k {
    use a::b::{F, G};
    // ... accesses to F and F.x ...
    // ... accesses to G and G.0 ...
}
```

Crate 限制示例

Crate `c1` :

```
pub mod a {
    struct Priv(i32);

    pub(crate) struct R { pub y: i32, z: Priv } // ok: field allowed to be more public
    pub        struct S { pub y: i32, z: Priv }

    pub fn to_r_bad(s: S) -> R { ... } //~ ERROR: `R` restricted solely to this crate

    pub(crate) fn to_r(s: S) -> R { R { y: s.y, z: s.z } } // ok: restricted to crate
}

use a::{R, S}; // ok: `a::R` and `a::S` are both visible

pub use a::R as ReexportAttempt; //~ ERROR: `a::R` restricted solely to this crate
```

Crate `c2`:

```
extern crate c1;

use c1::a::S; // ok: `S` is unrestricted

use c1::a::R; //~ ERROR: `c1::a::R` not visible outside of its crate
```

17.错误处理

错误处理是保证程序健壮性的前提，在编程语言中错误处理的方式大致分为两种：抛出异常（exceptions）和作为值返回。

Rust 将错误作为值返回并且提供了原生的优雅的错误处理方案。

熟练掌握错误处理是软件工程中非常重要的环节，让我一起来看看**Rust**展现给我们的错误处理艺术。

17.1 Option和Result

谨慎使用 `panic` ：

```
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!("Invalid number: {}", n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

`panic` 会导致当前线程结束，甚至是整个程序的结束，这往往是不被期望看到的结果。（编写示例或者简短代码的时候 `panic` 不失为一个好的建议）

Option

```
enum Option<T> {
    None,
    Some(T),
}
```

Option 是Rust的系统类型，用来表示值不存在的可能，这在编程中是一个好的实践，它强制**Rust**检测和处理值不存在的情况。例如：

```
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

`find` 在字符串 `haystack` 中查找 `needle` 字符，事实上结果会出现两种可能，有（ `Some(usize)`)或无（ `None` ）。

```
fn main() {
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("No file extension found."),
        Some(i) => println!("File extension: {}", &file_name[i+1
..]),
    }
}
```

Rust 使用模式匹配来处理返回值，调用者必须处理结果为 `None` 的情况。这往往是一个好的编程习惯，可以减少潜在的bug。**Option** 包含一些方法来简化模式匹配，毕竟过多的 `match` 会使代码变得臃肿，这也是滋生bug的原因之一。

unwrap

```
impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None` valu
e"),
        }
    }
}
```

`unwrap` 当遇到 `None` 值时会panic，如前面所说这不是一个好的工程实践。不过有些时候却非常有用：

- 在例子和简单快速的编码中 有的时候你只是需要一个小例子或者一个简单的小程序，输入输出已经确定，你根本没必要花太多时间考虑错误处理，使用 `unwrap` 变得非常合适。
- 当程序遇到了致命的**bug**，**panic**是最优选择

map

假如我们要在一个字符串中找到文件的扩展名，比如 `foo.rs` 中的 `rs` ，我们可以这样：

```
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}

fn main() {
    match extension_explicit("foo.rs") {
        None => println!("no extension"),
        Some(ext) =>  assert_eq!(ext, "rs"),
    }
}
```

我们可以使用 `map` 简化：

```
// map是标准库中的方法
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where F: F
nOnce(T) -> A {
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
// 使用map去掉match
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

`map` 如果有值 `Some(T)` 会执行 `f` ，反之直接返回 `None` 。

unwrap_or

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

`unwrap_or` 提供了一个默认值 `default` ，当值为 `None` 时返回 `default` ：

```
fn main() {
    assert_eq!(extension("foo.rs").unwrap_or("rs"), "rs");
    assert_eq!(extension("foo").unwrap_or("rs"), "rs");
}
```

and_then

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
    }
}
```

看起来 `and_then` 和 `map` 差不多，不过 `map` 只是把值为 `Some(t)` 重新映射了一遍，`and_then` 则会返回另一个 `Option`。如果我们在一个文件路径中找到它的扩展名，这时候就会变得尤为重要：

```
use std::path::Path;
fn file_name(file_path: &str) -> Option<&str> {
    let path = Path::new(file_path);
    path.file_name().to_str()
}
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` 是 `Option` 的更通用的版本，比起 `Option` 结果为 `None` 它解释了结果错误的原因，所以：

```
type Option<T> = Result<T, ()>;
```

这样的别名是一样的（`()` 标示空元组，它既是 `()` 类型也可以是 `()` 值）

unwrap

```
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err` valu
e: {:?}", err),
        }
    }
}
```

没错和 `Option` 一样，事实上它们拥有很多类似的方法，不同的是，`Result` 包括了错误的详细描述，这对于调试人员来说，这是友好的。

Result我们从例子开始

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

`double_number` 从一个字符串中解析出一个 `i32` 的数字并 `*2` ，`main` 中调用看起来没什么问题，但是如果把 `"10"` 换成其他解析不了的字符串程序便会panic

```
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

`parse` 返回一个 `Result` ，但让我们也可以返回一个 `Option` ，毕竟一个字符串要么能解析成一个数字要么不能，但是 `Result` 给我们提供了更多的信息（要么是一个空字符串，一个无效的数位，太大或太小），这对于使用者是友好的。当你面

对一个Option和Result之间的选择时。如果你可以提供详细的错误信息，那么大概你也应该提供。

这里需要理解一下 `FromStr` 这个**trait**:

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}

impl FromStr for i32 {
    type Err = ParseIntError;
    fn from_str(src: &str) -> Result<i32, ParseIntError> {

    }
}
```

`number_str.parse::<i32>()` 事实上调用的是 `i32` 的 `FromStr` 实现。

我们需要改写这个例子：

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError>
 {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

不仅仅是 `map` ，`Result` 同样包含了 `unwrap_or` 和 `and_then` 。也有一些特有的针对错误类型的方法 `map_err` 和 `or_else` 。

Result别名

在**Rust**的标准库中会经常出现Result的别名，用来默认确认其中 `Ok(T)` 或者 `Err(E)` 的类型，这能减少重复编码。比如 `io::Result`

```
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

组合Option和Result

`Option` 的方法 `ok_or` ：

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

可以在值为 `None` 的时候返回一个 `Result::Err(E)` ，值为 `Some(T)` 的时候返回 `Ok(T)` ，利用它我们可以组合 `Option` 和 `Result` ：

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_
string()))
        .map(|n| 2 * n)
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

`double_arg` 将传入的命令行参数转化为数字并翻倍，`ok_or` 将 `Option` 类型转换成 `Result` ，`map_err` 当值为 `Err(E)` 时调用作为参数的函数处理错误

复杂的例子

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Stri
ng> {
    File::open(file_path)
         .map_err(|err| err.to_string())
         .and_then(|mut file| {
              let mut contents = String::new();
              file.read_to_string(&mut contents)
                  .map_err(|err| err.to_string())
                  .map(|_| contents)
         })
         .and_then(|contents| {
              contents.trim().parse::<i32>()
                      .map_err(|err| err.to_string())
         })
         .map(|n| 2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

`file_double` 从文件中读取内容并将其转化成 `i32` 类型再翻倍。 这个例子看起来已经很复杂了，它使用了多个组合方法，我们可以使用传统的 `match` 和 `if let` 来改写它：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Stri
ng> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

这两种方法个人认为都是可以的，依具体情况来取舍。

try!

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

`try!` 事实上就是 `match Result` 的封装，当遇到 `Err(E)` 时会提早返回，`::std::convert::From::from(err)` 可以将不同的错误类型返回成最终需要的错误类型，因为所有的错误都能通过 `From` 转化成 `Box<Error>` ，所以下面的代码是正确的：

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Box<
Error>> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n = try!(contents.trim().parse::<i32>());
    Ok(2 * n)
}
```

组合自定义错误类型

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::io;
use std::path::Path;

// We derive `Debug` because all types should probably derive `D
ebug`.
// This gives us a reasonable human readable description of `Cli
```

```
Error` values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Result<i
32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::
Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io
));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::
Parse));
    Ok(2 * n)
}
```

`CliError` 分别为 `io::Error` 和 `num::ParseIntError` 实现了 `From` 这个 trait，所有调用 `try!` 的时候这两种错误类型都能转化成 `CliError` 。

总结

熟练使用 `Option` 和 `Result` 是编写 **Rust** 代码的关键，**Rust** 优雅的错误处理离不开值返回的错误形式，编写代码时提供给使用者详细的错误信息是值得推崇的。

输入与输出

输入与输出可以说是一个实用程序的最基本要求，没有输入输出的程序是没有什么卵用的。虽然输入输出被函数式编程语言鄙称为副作用，但正是这个副作用才赋予了程序实用性，君不见某著名函数式语言之父称他主导设计的函数式语言"is useless"。这章我们就来谈谈输入输出副作用。

读写 Trait

输入最基本的功能是读(Read)，输出最基本的功能是写(Write)。标准库里面把怎么读和怎么写抽象出来归到了 `Read` 和 `Write` 两个接口里面，实现了 `Read` 接口的叫 reader，而实现了 `Write` 的叫 writer。Rust里面的 Trait 比其它语言里面的接口更好的一个地方是 Trait 可以带默认实现，比如用户定义的 reader 只需要实现 `read` 一个方法就可以调用 `Read` trait 里面的任意其它方法，而 writer 也只需要实现 `write` 和 `flush` 两个方法。

Read 和 Write 这两个 Trait 都有定义了好多方法，具体可以参考标准库 API 文档中的Read 和 Write

Read 由于每调用一次 `read` 方法都会调用一次系统API与内核交互，效率比较低，如果给 reader 增加一个 buffer，在调用时 `read` 方法时多读一些数据放在 buffer 里面，下次调用 `read` 方法时就有可能只需要从 buffer 里面取数据而不用调用系统API了，从而减少了系统调用次数提高了读取效率，这就是所谓的 `BufRead` Trait。一个普通的 reader 通过 `io::BufReader::new(reader)` 或者 `io::BufReader::with_capacity(bufSize, reader)` 就可以得到一个 BufReader 了，显然这两个创建 BufReader 的函数一个是使用默认大小的 buffer 一个可以指定 buffer 大小。BufReader 比较常用的两个方法是按行读：`read_line(&mut self, buf: &mut String) -> Result<usize>` 和 `lines(&mut self) -> Lines<Self>`，从函数签名上就可以大概猜出函数的用法所以就不啰嗦了，需要注意的是后者返回的是一个迭代器。详细说明直接看 API 文档中的BufRead

同样有 `BufWriter` 只不过由于其除了底层加了 buffer 之外并没有增加新的写方法，所以并没有专门的 `BufWrite` Trait，可以通过 `io::BufWriter::new(writer)` 或

`io::BufWriter::with_capacity(bufSize, writer)` 创建 `BufWriter` 。

输入与输出接口有了，我们接下来看看实际应用中最常用的两类 reader 和 writer：标准输入/输出，文件输入/输出

标准输入与输出

回顾一下我们写的第一个 Rust 程序就是带副作用的，其副作用就是向标准输出(stdout)，通常是终端或屏幕，输出了 Hello, World! 让屏幕上这几个字符的地方点亮起来。 `println!` 宏是最常见的输出，用宏来做输出的还有 `print!` ，两者都是向标准输出(stdout)输出，两者的区别也一眼就能看出。至于格式化输出，基础运算符和字符串格式化小节有详细说明，这里就不再啰嗦了。

更通用的标准输入与输出定义在 `std::io` 模块里，调用 `std::io::stdin()` 和 `std::io::stdout()` 两个函数分别会得到输入句柄和输出句柄(哎，句柄这个词是计算机史上最莫名其妙的翻译了)，这两个句柄默认会通过互斥锁同步，也就是说不让多个进程同时读或写标准输入输出，不然的话如果一个进程要往标准输出画马，一个进程要画驴，两个进程同时写标准输出的话，最后可能就给画出一头骡子了，如果更多进程画不同的动物最后可能就成四不像了。除了隐式地用互斥锁，我们还可以显式地在句柄上调用 `.lock()` 。输入输出句柄实现了前面讲的读写 Trait，所以是 reader/writer，就可以调接口来读写标准输入与输出了。举几个栗子：

```
use std::io;

fn read_from_stdin(buf: &mut String) -> io::Result<()> {
    try!(io::stdin().read_line(buf));
    Ok(())
}
```

```
use std::io;

fn write_to_stdout(buf: &[u8]) -> io::Result<()> {
    try!(io::stdout().write(&buf));
    Ok(())
}
```

可以看到上面的例子都是返回了 `io::Result<()>` 类型，这不是偶然，而是 IO 操作通用的写法，因为 IO 操作是程序与外界打交道，所以都是有可能失败的，用 `io::Result<T>` 把结果包起来， `io::Result<T>` 只是标准 `Result<T,E>` 中

`E` 固定为 `io::Error` 后类型的别名，而作为有副作用的操作我们一般是不用关心其返回值的，因为执行这类函数其真正的意义都体现在副作用上面了，所以返回值只是用来表示是否成功执行，而本身 `Result` 类型本身已经可以表示执行状态了，里面的 `T` 是什么则无关紧要，既然 `T` 没什么意义，那我们就选没什么意义的 `unit` 类型好了，所以 IO 操作基本上都是使用 `io::Result<()>` 。

另外有一个地方需要注意的是由于 IO 操作可能会失败所以一般都是和 `try!` 宏一起使用的，但是 `try!` 在遇到错误时会把错误 `return` 出去的，所以需要保证包含 `try!` 语句的函数其返回类型是 `io::Result<T>` ，很多新手文档没仔细看就直接查 std api 文档，然后照着 api 文档里面的例子把带 IO 操作的 `try!` 宏写到了 `main` 函数里。结果一编译，擦，照着文档写都编译不过，什么烂文档。其实点一下 api 文档上面的运行按钮就会发现文档里面的例子都是把 `try!` 放在另一个函数里面的，因为 `main` 函数是没有返回值的，而 `try!` 会返回 `io::Result<T>` ，所以直接把 `try!` 放 `main` 函数里面肯定要跪。比如下面的从标准输入读取一行输入，由于把 `try!` 放在了 main 函数里，所以是编译不过的。

```rust
use std::io;

fn main() {
    let mut input = String::new();
    try!(io::stdin().read_line(&mut input));
    println!("You typed: {}", input.trim());
}
```

这里有一件事需要主要的是 Rust 里面没有办法从键盘获取一个数字类型的值。实际上像 C 这样的语言也不是直接获取了数字类型，它只不过是做了一种转换。那么我们如果想要从键盘获取一个数字类型应该怎么做呢？

```
fn main() {
    let mut input = String::new();
        std::io::stdin()
            .read_line(&mut input)
            .expect("Failed to read line");
    // 这里等效的写法是：
    // let num: i32 = input.trim().parse().unwrap();
    let num = input.trim().parse::<i32>().unwrap();
    println!("您输入的数字是：{}", num);
}
```

如果有很多地方都需要输入数字可以自行编写一个 `numin` 宏:

```
macro_rules! numin {
      () =>{
          {
            let mut input = String::new();
              std::io::stdin()
                  .read_line(&mut input)
                .expect("Failed to read line");
              input.trim().parse().unwrap()
        }
    };
}
```

于是上面的程序可以被改写成：

```

fn main() {
    let num: i32 = numin!();
    println!("您输入的数字是：{}", num);
}
```

不过如果用户输入的不是数字，那么就会导致错误。这一点和 C 里面是非常相似的。当然您可以把程序写得再复杂一点儿来保证用户输入的一定是数字。不过这些就不是我们这一节要讨论的内容了。

还有一点一些从其它语言转过来的程序员可能会疑惑的是，如何从命令行接受输入参数，因为 C 里面的 main 函数可以带参数所以可以直接从 main 函数的参数里获取输入参数。但其实这类输入与我们这里讲的有很大的差别的，它在 Rust 里面被归为环境变量，可以通过 `std::env::args()` 获取，这个函数返回一个 `Args` 迭代器，其中第一个就是程序名，后面的都是输入给程序的命令行参数。

```
use std::env;

fn main() {
    let args = env::args();
    for arg in args {
        println!("{}", arg);
    }
}
```

将上面的程序存为 *args.rs* 然后编译执行，结果如下

```
$ rustc args.rs
$ ./args a b c
./args
a
b
c
```

print! 宏

我们在快速入门中就提到过标准输出的行缓冲。它一个表现就是 `print!` 宏。如果你在 `print!` 宏后面接上一个输入就会发现这种按行缓冲的机制。

```
fn main() {
    print!("hello!\ninput:");
    let mut input = String::new();
        std::io::stdin()
            .read_line(&mut input)
            .expect("Failed to read line");
    println!("line:{}",input);
}
```

您可以编译并运行这段程序试一试，您会发现我们并没有得到预期的（下划线代表光标的位置）：

```
hello!
input:_
```

而是得到了：

```
hello!
_
```

这就是由于标准输出中的这种行缓冲机制，在遇到换行符之前，输出的内容并不会隐式的刷新，这就导致 `print!` 宏和 `println!` 宏实际上并不完全相同。在标准库中 `print!` 宏是这样的：

```
macro_rules! print {
    ($($arg:tt)*) => { ... };
}
```

由此，我们可以对它进行改进，使它和 `println!` 宏被自动刷新，不过这种刷新是一种显式的刷新。

```
use std::io::{self, Write};

macro_rules! printf {
    ($($arg:tt)*) =>{
        print!($($arg)*);
        io::stdout().flush().unwrap();
    }
}
```

此外，当您需要刷新还没有遇到换行符的一行内容的时候您都可以使用 `io::stdout().flush().unwrap();` 进行刷新，不过需要注意的是要先 `use std::io::{self, Write};` 如果您不这样做，将会得到一个错误。

文件输入与输出

文件 `std::fs::File` 本身实现了 `Read` 和 `Write` trait，所以文件的输入输出非常简单，只要得到一个 `File` 类型实例就可以调用读写接口进行文件输入与输出操作了。而要得到 `File` 就得让操作系统打开(open)或新建(create)一个文件。还是拿例子来说明

```
use std::io;
use std::io::prelude::*;
use std::fs::File;

// create file and write something
fn create_file(filename: &str, buf: &[u8]) -> io::Result<()> {
    let mut f = try!(File::create(filename));
    try!(f.write(&buf));
    Ok(())
}

// read from file to String
fn read_file(filename: &str, buf: &mut String) -> io::Result<()>
 {
    let mut f = try!(File::open(filename));
    try!(f.read_to_string(&buf));
    Ok(())
}

fn main() {
    let f = "foo.txt";
    let mut buf = String::new();
    match create_file(f, b"Hello, World!") {
        Ok(()) => {
            match read_file(f, &mut buf) {
                Ok(()) => {println!("{}", buf);},
                Err(e) => {println!("{}", e);},
            };
        },
        Err(e) => {println!("{}", e);},
    }
}
```

文件操作上面 Rust 与其它语言处理方式有些不一样，其它语言一般把读写选项作为函数参数传给 open 函数，而 Rust 则是在 option 上面调用 open 函数。

`std::fs::OpenOptions` 是一个 builder，通过 new 函数创建后，可以链式调用

设置打开文件的选项，是 read, write, append, truncate 还是 create 等，OpenOptions 构建完成后就可以再接着调用 open 方法了，看下下面的例子就明白了

```
use std::fs::OpenOptions;

let file = OpenOptions::new().write(true).truncate(true).open("f
oo.txt");
```

Rust 这种用 builder pattern 来设置打开文件选项，相比于将选项以字符作为参数传给 open 函数的一个优点是可以让编译器保证检查选项合法性，不用等到运行时才发现手抖把 read-mode 的 `r` 写成了 `t` 。

Macro

简介

学过 C 语言的人都知道 `#define` 用来定义宏(macro)，而且大学很多老师都告诉你尽量少用宏，因为 C 里面的宏是一个很危险的东西-宏仅仅是简单的文本替换，完全不管语法，类型，非常容易出错。听说过或用过 Lisp 的人觉得宏极其强大，就连美国最大的创业孵化器公司创始人 Paul Gram 也极力鼓吹 Lisp 的宏是有多么强大。那么宏究竟是什么样的东西呢？这一章通过 Rust 的宏系统带你揭开宏(Macro)的神秘面纱。

Rust 中的宏几乎无处不在，其实你写的第一个 Rust 程序里面就已经用到了宏，对，就是那个有名的 hello-world。`println!("Hello, world!")` 这句看起来很像函数调用，但是在"函数名"后面加上了感叹号，这个是专门用来区分普通函数调用和宏调用的。另外从形式上看，与函数调用的另一个区别是参数可以用圆括号(`()`)、花括号(`{}`)、方括号(`[]`)中的任意一种括起来，比如这行也可以写成 `println!["Hello, world!"]` 或 `println!{"Hello, world!"}` ，不过对于 Rust 内置的宏都有约定俗成的括号，比如 `vec!` 用方括号， `assert_eq!` 用圆括号。

既然宏看起来与普通函数非常像，那么使用宏有什么好处呢？是否可以用函数取代宏呢？答案显然是否定的，首先 Rust 的函数不能接受任意多个参数，其次函数是不能操作语法单元的，即把语法元素作为参数进行操作，从而生成代码，例如 `mod` , `crate` 这些是 Rust 内置的关键词，是不可能直接用函数去操作这些的，而宏就有这个能力。

相比函数，宏是用来生成代码的，在调用宏的地方，编译器会先将宏进行展开，生成代码，然后再编译展开后的代码。

宏定义格式是： `macro_rules! macro_name { macro_body }` ，其中 `macro_body` 与模式匹配很像， `pattern => do_something` ，所以 Rust 的宏又称为 Macro by example (基于例子的宏)。其中 `pattern` 和 `do_something` 都是用配对的括号括起来的，括号可以是圆括号、方括号、花括号中的任意一种。匹配可以有多个分支，每个分支以分号结束。

还是先来个简单的例子说明

```
macro_rules! create_function {
    ($func_name:ident) => (
        fn $func_name() {
            println!("function {:?} is called", stringify!($func
_name))
        }
    )
}

fn main() {
    create_function!(foo);
    foo();
}
```

上面这个简单的例子是用来创建函数，生成的函数可以像普通函数一样调用，这个函数可以打印自己的名字。编译器在看到 `create_function!(foo)` 时会从前面去找一个叫 `create_function` 的宏定义，找到之后，就会尝试将参数 `foo` 代入 `macro_body` ，对每一条模式按顺序进行匹配，只要有一个匹配上，就会将 `=>` 左边定义的参数代入右边进行替换，如果替换不成功，编译器就会报错而不会往下继续匹配，替换成功就会将右边替换后的代码放在宏调用的地方。这个例子中只有一个模式，即 `$func_name:ident` ，表示匹配一个标识符，如果匹配上就把这个标识符赋值给 `$func_name` ，宏定义里面的变量都是以 `$` 开头的，相应的类型也是以冒号分隔说明，这里 `ident` 是变量 `$func_name` 的类型，表示这个变量是一个 `identifier` ，这是语法层面的类型(designator)，而普通的类型如 `char, &str, i32, f64` 这些是语义层面的类型。在 `main` 函数中传给宏调用 `create_function` 的参数 `foo` 正好是一个标识符(`ident`)，所以能匹配上， `$func_name` 就等于 `foo` ，然后把 `$func_name` 的值代入 `=>` 右边，成了下面这样的

```
fn foo() {
    println!("function {:?} is called", stringify!(foo))
}
```

所以最后编译器编译的实际代码是

```
fn main() {
    fn foo() {
        println!("function {:?} is called", stringify!(foo))
    }
    foo();
}
```

上面定义了 `create_function` 这个宏之后，就可以随便用来生成函数了，比如调用 `create_function!(bar)` 就得到了一个名为 `bar` 的函数

通过上面这个例子，大家对宏应该有一个大致的了解了。下面就具体谈谈宏的各个组成部分。

宏的结构

宏名

宏名字的解析与函数略微有些不同，宏的定义必须出现在宏调用之前，即与 C 里面的函数类似--函数定义或声明必须在函数调用之前，只不过 Rust 宏没有单纯的声明，所以宏在调用之前需要先定义，而 Rust 函数则可以定义在函数调用后面。宏调用与宏定义顺序相关性包括从其它模块中引入的宏，所以引入其它模块中的宏时要特别小心，这个稍后会详细讨论。

下面这个例子宏定义在宏调用后面，编译器会报错说找不到宏定义，而函数则没问题

```
fn main() {
    let a = 42;
    foo(a);
    bar!(a);
}

fn foo(x: i32) {
    println!("The argument you passed to function is {}", x);
}

macro_rules! bar {
    ($x:ident) => { println!("The argument you passed to macro is {}", $x); }
}
```

上面例子中把宏定义挪到 `main` 函数之前或者 `main` 函数里面 `bar!(a)` 调用上面，就可以正常编译运行。

宏调用虽然与函数调用很像，但是宏的名字与函数名字是处于不同命名空间的，之所以提出来是因为在有些编程语言里面宏和函数是在同一个命名空间之下的。看过下面的例子就会明白

```
fn foo(x: i32) -> i32 {
    x * x
}

macro_rules! foo {
    ($x:ident) => { println!("{:?}", $x); }
}
fn main() {
    let a = 5;
    foo!(a);
    println!("{}", foo(a));
}
```

指示符(designator)

宏里面的变量都是以 `$` 开头的，其余的都是按字面去匹配，以 `$` 开头的变量都是用来表示语法(syntactic)元素，为了限定匹配什么类型的语法元素，需要用指示符(designator)加以限定，就跟普通的变量绑定一样用冒号将变量和类型分开，当前宏支持以下几种指示符：

- ident: 标识符，用来表示函数或变量名
- expr: 表达式
- block: 代码块，用花括号包起来的多个语句
- pat: 模式，普通模式匹配（非宏本身的模式）中的模式，例如 `Some(t)` , `(3, 'a', _)`
- path: 路径，注意这里不是操作系统中的文件路径，而是用双冒号分隔的限定名(qualified name)，如 `std::cmp::PartialOrd`
- tt: 单个语法树
- ty: 类型，语义层面的类型，如 `i32` , `char`
- item: 条目，
- meta: 元条目
- stmt: 单条语句，如 `let a = 42;`

加上这些类型限定后，宏在进行匹配时才不会漫无目的的乱匹配，例如在要求标识符的地方是不允许出现表达式的，否则编译器就会报错。而 C/C++ 语言中的宏则仅仅是简单的文本替换，没有语法层面的考虑，所以非常容易出错。

重复(repetition)

宏相比函数一个很大的不同是宏可以接受任意多个参数，例如 `println!` 和 `vec!` 。这是怎么做到的呢？

没错，就是重复(repetition)。模式的重复不是通过程序里面的循环(for/while)去控制的，而是指定了两个特殊符号 `+` 和 `*` ，类似于正则表达式，因为正则表达式也是不关心具体匹配对象是一个人名还是一个国家名。与正则表达式一样， `+` 表示一次或多次（至少一次），而 `*` 表示零次或多次。重复的模式需要用括号括起来，外面再加上 `$` ，例如 `$(...)*` , `$(...)+` 。需要说明的是这里的括号和宏里面其它地方一样都可以是三种括号中的任意一种，因为括号在这里仅仅是用来标记一个模式的开始和结束，大部分情况重复的模式是用逗号或分号分隔的，所以你会经常看到 `$(...),*` , `$(...);*` , `$(...),+` , `$(...);+` 这样的用来表示重复。

还是来看一个例子

```
macro_rules! vector {
    ($($x:expr),*) => {
        {
            let mut temp_vec = Vec::new();
            $(temp_vec.push($x);)*
            temp_vec
        }
    };
}

fn main() {
    let a = vector![1, 2, 4, 8];
    println!("{:?}", a);
}
```

这个例子初看起来比较复杂，我们来分析一下。

首先看 `=>` 左边，最外层是圆括号，前面说过这个括号可以是圆括号、方括号、花括号中的任意一种，只要是配对的就行。然后再看括号里面 `$(...),*` 正是刚才提到的重复模式，重复的模式是用逗号分隔的，重复的内容是 `$x:expr` ，即可以匹配零次或多次用逗号分隔的表达式，例如 `vector![]` 和 `vector![3, x*x, s-t]` 都可以匹配成功。

接着看 `=>` 右边，最外层也是一个括号，末尾是分号表示这个分支结束。里面是花括号包起来的代码块，最后一行没有分号，说明这个 macro 的值是一个表达式， `temp_vec` 作为表达式的值返回。第一条语句就是普通的用 `Vec::new()` 生成一个空 vector，然后绑定到可变的变量 `temp_vec` 上面，第二句比较特殊，跟 `=>` 左边差不多，也是用来表示重复的模式，而且是跟左边是一一对应的，即左边匹配到一个表达式(`expr`)，这里就会将匹配到的表达式用在 `temp_vec.push($x);` 里面，所以 `vector![3, x*x, s-t]` 调用就会展开成

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(3);
    temp_vec.push(x*x);
    temp_vec.push(s-t);
    temp_vec
}
```

看着很复杂的宏，细细分析下来是不是很简单，不要被这些符号干扰了

递归(recursion)

除了重复之外，宏还支持递归，即在宏定义时调用其自身，类似于递归函数。因为rust的宏本身是一种模式匹配，而模式匹配里面包含递归则是函数式语言里面最常见的写法了，有函数式编程经验的对这个应该很熟悉。下面看一个简单的例子：

```
macro_rules! find_min {
    ($x:expr) => ($x);
    ($x:expr, $($y:expr),+) => (
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1u32));
    println!("{}", find_min!(1u32 + 2 , 2u32));
    println!("{}", find_min!(5u32, 2u32 * 3, 4u32));
}
```

因为模式匹配是按分支顺序匹配的，一旦匹配成功就不会再往下进行匹配（即使后面也能匹配上），所以模式匹配中的递归都是在第一个分支里写最简单情况，越往下包含的情况越多。这里也是一样，第一个分支 `($x:expr)` 只匹配一个表达式，第二个分支匹配两个或两个以上表达式，注意加号表示匹配一个或多个，然后里面是用标准库中的 `min` 比较两个数的大小，第一个表达式和剩余表达式中最小的一个，其中剩余表达式中最小的一个是递归调用 `find_min!` 宏，与递归函数一样，每次递归都是从上往下匹配，只到匹配到基本情况。我们来写写 `find_min!(5u32, 2u32 * 3, 4u32)` 宏展开过程

1. `std::cmp::min(5u32, find_min!(2u32 * 3, 4u32))`
2. `std::cmp::min(5u32, std::cmp::min(2u32 * 3, find_min!(4u32)))`
3. `std::cmp::min(5u32, std::cmp::min(2u32 * 3, 4u32))`

分析起来与递归函数一样，也比较简单。

卫生(hygienic Macro)

有了重复和递归，组合起来就是一个很强大的武器，可以解决很多普通函数无法抽象的东西。但是这里面会有一个安全问题，也是 C/C++ 里面宏最容易出错的地方，不过 Rust 像 Scheme 一样引入了卫生(Hygiene)宏，有效地避免了这类问题的发生。

C/C++ 里面的宏仅仅是简单的文本替换，下面的 C 经过宏预处理后，宏外面定义的变量 `a` 就会与里面定义的混在一起，从而按作用域 shadow 外层的定义，这会导致一些非常诡异的问题，不去看宏具体定义仔细分析的话，很难发现这类 bug。这样的宏是不卫生的，不过也有些奇葩的 Hacker 觉得这是一个非常棒的特性，例如 CommanLisp 语言里面的宏本身很强大，但不是卫生的，而某些 Hacker 还以这个为傲，搞一些奇技淫巧故意制造出这样的 shadow 行为实现一些很 fancy 的效果。这里不做过多评论，对 C 比较熟悉的同学可以分析一下下面这段代码运行结果与第一印象是否一样。

```
#define INCI(i) {int a=0; ++i;}
int main(void)
{
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

卫生宏最开始是由 Scheme 语言引入的，后来好多语言基本都采用卫生宏，即编译器或运行时会保证宏里面定义的变量或函数不会与外面的冲突，在宏里面以普通方式定义的变量作用域不会跑到宏外面。

```
macro_rules! foo {
    () => (let x = 3);
}

macro_rules! bar {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!();
    println!("{}", x);
    bar!(a);
    println!("{}", a);
}
```

上面代码中宏 `foo!` 里面的变量 `x` 是按普通方式定义的，所以其作用域限定在宏里面，宏调用结束后再引用 `x` 编译器就会报错。要想让宏里面定义的变量在宏调用结束后仍然有效，需要按 `bar!` 里面那样定义。不过对于 `item` 规则就有些不同，例如函数在宏里面以普通方式定义后，宏调用之后，这个函数依然可用，下面代码就可以正常编译。

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

导入导出(import/export)

前面提到宏名是按顺序解析的，所以从其它模块中导入宏时与导入函数、trait 的方式不太一样，宏导入导出用 `#[macro_use]` 和 `#[macro_export]` 。父模块中定义的宏对其下的子模块是可见的，要想子模块中定义的宏在其后面的父模块中可用，需要使用 `#[macro_use]` 。

```
macro_rules! m1 { () => (()) }

// 宏 m1 在这里可用

mod foo {
    // 宏 m1 在这里可用

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // 宏 m1 和 m2 在这里可用
}

// 宏 m1 在这里可用
#[macro_export]
macro_rules! m3 { () => (()) }

// 宏 m1 和 m3 在这里可用

#[macro_use]
mod bar {
    // 宏 m1 和 m3 在这里可用

    macro_rules! m4 { () => (()) }

    // 宏 m1, m3, m4 在这里均可用
}

// 宏 m1, m3, m4 均可用
```

crate 之间只有被标为 `#[macro_export]` 的宏可以被其它 crate 导入。假设上面例子是 `foo` crate 中的部分代码，则只有 `m2` 和 `m3` 可以被其它 crate 导入。导入方式是在 `extern crate foo;` 前面加上 `#[macro_use]`

```
#[macro_use]
extern crate foo;
// foo 中 m2, m3 都被导入
```

如果只想导入 `foo` crate 中某个宏，比如 `m3` ，就给 `#[macro_use]` 加上参数

```
#[macro_use(m3)]
extern crate foo;
// foo 中只有 m3 被导入
```

调试

虽然宏功能很强大，但是调试起来要比普通代码困难，因为编译器默认情况下给出的提示都是对宏展开之后的，而不是你写的原程序，要想在编译器错误与原程序之间建立联系比较困难，因为这要求你大脑能够人肉编译展开宏代码。不过还好编译器为我们提供了 `--pretty=expanded` 选项，能让我们看到展开后的代码，通过这个展开后的代码，往上靠就与你自己写的原程序有个直接对应关系，往下靠与编译器给出的错误也是直接对应关系。

目前将宏展开需要使用 unstable option，通过 `rustc -Z unstable-options --pretty=expanded hello.rs` 可以查看宏展开后的代码，如果是使用的 cargo 则通过 `cargo rustc -- -Z unstable-options --pretty=expanded` 将项目里面的宏都展开。不过目前是没法只展开部分宏的，而且由于 hygiene 的原因，会对宏里面的名字做些特殊的处理(mangle)，所以程序里面的宏全部展开后代码的可读性比较差，不过依然比依靠大脑展开靠谱。

下面可以看看最简单的 hello-word 程序里面的 `println!("Hello, world!")` 展开结果，为了 hygiene 这里内部临时变量用了 `__STATIC_FMTSTR` 这样的名字以避免名字冲突，即使这简单的一句展开后看起来也还是不那么直观的，具体这里就不详细分析了。

```
$ rustc -Z unstable-options --pretty expanded hello.rs
#![feature(prelude_import)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
fn main() {
    ::std::io::_print(::std::fmt::Arguments::new_v1({
                                                        static _
_STATIC_FMTSTR:
                                                               &
'static [&'static str]
                                                               =
                                                            &["H
ello, world!\n"];
                                                        __STATIC
_FMTSTR
                                                    },
                                                    &match () {
() => [], }));
}
```

Heap & Stack

简介

堆和栈是计算机里面最基本的概念，不过如果一直使用高级语言如 Python/Ruby/PHP/Java 等之类的语言的话，可能对堆和栈并不怎么理解，当然这里的栈(Stack)并不是数据结构里面的概念，而是计算机对内存的一个抽象。相比而言，C/C++/Rust 这些语言就必须对堆和栈的概念非常了解才能写出正确的程序，之所以有这样的区别是因为它们的内存管理方式不同，Python 之类的语言程序运行时会同时会运行垃圾回收器，垃圾回收器与用户程序或并行执行或交错执行，垃圾回收器会自动释放不再使用的内存空间，而 C/C++/Rust 则没有垃圾回收器。

操作系统会将物理内存映射成虚拟地址空间，程序在启动时看到的虚拟地址空间是一块完整连续的内存。

栈内存从高位地址向下增长，且栈内存分配是连续的，一般操作系统对栈内存大小是有限制的，Linux/Unix 类系统上面可以通过 ulimit 设置最大栈空间大小，所以 C 语言中无法创建任意长度的数组。在Rust里，函数调用时会创建一个临时栈空间，调用结束后 Rust 会让这个栈空间里的对象自动进入 `Drop` 流程，最后栈顶指针自动移动到上一个调用栈顶，无需程序员手动干预，因而栈内存申请和释放是非常高效的。

相对地，堆上内存则是从低位地址向上增长，堆内存通常只受物理内存限制，而且通常是不连续的，一般由程序员手动申请和释放的，如果想申请一块连续内存，则操作系统需要在堆中查找一块未使用的满足大小的连续内存空间，故其效率比栈要低很多，尤其是堆上如果有大量不连续内存时。另外内存使用完也必须由程序员手动释放，不然就会出现内存泄漏，内存泄漏对需要长时间运行的程序(例如守护进程)影响非常大。

Rust 中的堆和栈

由于函数栈在函数执行完后会销毁，所以栈上存储的变量不能在函数之间传递，这也意味着函数没法返回栈上变量的引用，而这通常是 C/C++ 新手常犯的错误。而 Rust 中编译器则会检查出这种错误，错误提示一般为 `xxx does not live long enough` ，看下面一个例子

```
fn main() {
    let b = foo("world");
    println!("{}", b);
}

fn foo(x: &str) -> &str {
    let a = "Hello, ".to_string() + x;
    &a
}
```

之所以这样写，很多人觉得可以直接拷贝字符串 `a` 的引用从而避免拷贝整个字符串，然而得到的结果却是 `a does not live long enough` 的编译错误。因为引用了一个函数栈中临时创建的变量，函数栈在函数调用结束后会销毁，这样返回的引用就变得毫无意义了，指向了一个并不存在的变量。相对于 C/C++ 而言，使用 Rust 就会幸运很多，因为 C/C++ 中写出上面那样的程序，编译器会默默地让你通过直到运行时才会给你报错。

其实由于 `a` 本身是 String 类型，是使用堆来存储的，所以可以直接返回，在函数返回时函数栈销毁后依然存在。同时 Rust 中下面的代码实际上也只是浅拷贝。

```
fn main() {
    let b = foo("world");
    println!("{}", b);
}

fn foo(x: &str) -> String {
    let a = "Hello, ".to_string() + x;
    a
}
```

Rust 默认使用栈来存储变量，而栈上内存分配是连续的，所以必须在编译之前了解变量占用的内存空间大小，编译器才能合理安排内存布局。

Box

C 里面是通过 malloc/free 手动管理堆上内存空间的，而 Rust 则有多种方式，其中最常用的一种就是 Box,通过 `Box::new()` 可以在堆上申请一块内存空间，不像 C 里面一样堆上空间需要手动调用 `free` 释放，Rust 中是在编译期编译器借助 lifetime 对堆内存生命期进行分析，在生命期结束时自动插入 `free` 。当前 Rust 底层即 Box 背后是调用 jemalloc 来做内存管理的，所以堆上空间是不需要程序员手动去管理释放的。很多时候你被编译器虐得死去活来时，那些 `borrow` , `move` , `lifetime` 错误其实就是编译器在教你认识内存布局，教你用 lifetime 规则去控制内存。这套规则说难不难，说简单也不简单，以前用别的语言写程序时对内存不关心的，刚写起来可能真的会被虐得死去活来，但是一旦熟悉这套规则，对内存布局掌握清楚后，借助编译器的提示写起程序来就会如鱼得水，这套规则是理论界研究的成果在Rust编译器上的实践。

大多数带 GC 的面向对象语言里面的对象都是借助 box 来实现的，比如常见的动态语言 Python/Ruby/JavaScript 等，其宣称的"一切皆对象(Everything is an object)"，里面所谓的对象基本上都是 boxed value。

boxed 值相对于 unboxed，内存占用空间会大些，同时访问值的时候也需要先进行 unbox，即对指针进行解引用再获取真正存储的值，所以内存访问开销也会大些。既然 boxed 值既费空间又费时间，为什么还要这么做呢？因为通过 box，所有对象看起来就像是以相同大小存储的，因为只需要存储一个指针就够了，应用程序可以同等看待各种值，而不用去管实际存储是多大的值，如何申请和释放相应资源。

Box 是堆上分配的内存，通过 `Box::new()` 会创建一个堆空间并返回一个指向堆空间的指针

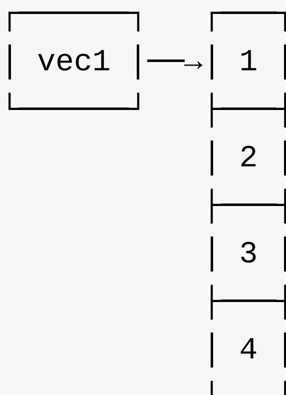
nightly 版本中引入 `box` 关键词，可以用来取代 `Box::new()` 申请一个堆空间，也可以用在模式匹配上面

```
#![feature(box_syntax, box_patterns)]
fn main() {
    let boxed = Some(box 5);
    match boxed {
        Some(box unboxed) => println!("Some {}", unboxed),
        None => println!("None"),
    }
}
```

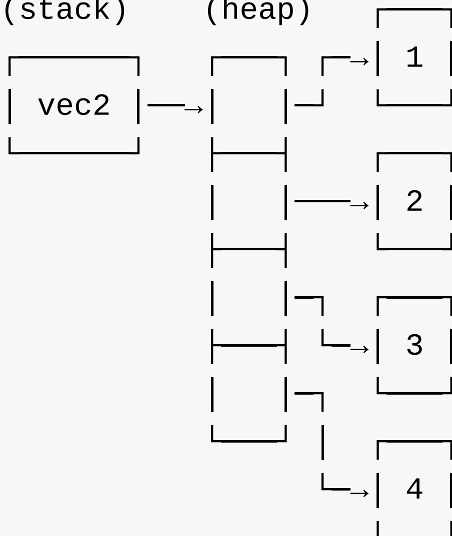
下面看一个例子，对比一下 `Vec<i32>` 和 `Vec<Box<i32>>` 内存布局，这两个图来自 Stack Overflow，从这两张内存分布图可以清楚直观地看出 Box 是如何存储的

```
Vec<i32>

(stack)    (heap)
```




```
Vec<Box<i32>>
```



一些语言里会有看起来既像数组又像列表的数据结构，例如 python 中的 List，其实就是与 `Vec<Box<i32>>` 类似，只是把 i32 换成任意类型，就操作效率而言比单纯的 List 高效，同时又比数组使用更灵活。

一般而言，在编译期间不能确定大小的数据类型都需要使用堆上内存，因为编译器无法在栈上分配 编译期未知大小 的内存，所以诸如 String, Vec 这些类型的内存其实是被分配在堆上的。换句话说，我们可以很轻松的将一个 Vec move 出作用域而不必担心消耗，因为数据实际上不会被复制。

另外,需要从函数中返回一个浅拷贝的变量时也需要使用堆内存而不能直接返回一个指向函数内部定义变量的引用。

几种智能指针

本章讲解 `Rc`，`Arc`，`Mutex`，`RwLock`，`Cell`，`RefCell` 的知识和使用方法。

Rc 和 Arc

Rust 建立在所有权之上的这一套机制，它要求一个资源同一时刻有且只能有一个拥有所有权的绑定或 `&mut` 引用，这在大部分的情况下保证了内存的安全。但是这样的设计是相当严格的，在另外一些情况下，它限制了程序的书写，无法实现某些功能。因此，Rust 在 std 库中提供了额外的措施来补充所有权机制，以应对更广泛的场景。

默认 Rust 中，对一个资源，同一时刻，有且只有一个所有权拥有者。 `Rc` 和 `Arc` 使用引用计数的方法，让程序在同一时刻，实现同一资源的多个所有权拥有者，多个拥有者共享资源。

Rc

`Rc` 用于同一线程内部，通过 `use std::rc::Rc` 来引入。它有以下几个特点：

1. 用 `Rc` 包装起来的类型对象，是 `immutable` 的，即 不可变的。即你无法修改 `Rc<T>` 中的 `T` 对象，只能读；
2. 一旦最后一个拥有者消失，则资源会被自动回收，这个生命周期是在编译期就确定下来的；
3. `Rc` 只能用于同一线程内部，不能用于线程之间的对象共享（不能跨线程传递）；
4. `Rc` 实际上是一个指针，它不影响包裹对象的方法调用形式（即不存在先解开包裹再调用值这一说）。

例子：

```
use std::rc::Rc;

let five = Rc::new(5);
let five2 = five.clone();
let five3 = five.clone();
```

Rc Weak

`Weak` 通过 `use std::rc::Weak` 来引入。

`Rc` 是一个引用计数指针，而 `Weak` 是一个指针，但不增加引用计数，是 `Rc` 的 weak 版。它有以下几个特点：

1. 可访问，但不拥有。不增加引用计数，因此，不会对资源回收管理造成影响；
2. 可由 `Rc<T>` 调用 `downgrade` 方法而转换成 `Weak<T>` ；
3. `Weak<T>` 可以使用 `upgrade` 方法转换成 `Option<Rc<T>>` ，如果资源已经被释放，则 Option 值为 `None` ；
4. 常用于解决循环引用的问题。

例子：

```
use std::rc::Rc;

let five = Rc::new(5);

let weak_five = Rc::downgrade(&five);

let strong_five: Option<Rc<_>> = weak_five.upgrade();
```

Arc

`Arc` 是原子引用计数，是 `Rc` 的多线程版本。 `Arc` 通过 `std::sync::Arc` 引入。

它的特点：

1. `Arc` 可跨线程传递，用于跨线程共享一个对象；
2. 用 `Arc` 包裹起来的类型对象，对可变性没有要求；
3. 一旦最后一个拥有者消失，则资源会被自动回收，这个生命周期是在编译期就确定下来的；
4. `Arc` 实际上是一个指针，它不影响包裹对象的方法调用形式（即不存在先解开包裹再调用值这一说）；
5. `Arc` 对于多线程的共享状态几乎是必须的（减少复制，提高性能）。

示例：

```
use std::sync::Arc;
use std::thread;

fn main() {
    let numbers: Vec<_> = (0..100u32).collect();
    let shared_numbers = Arc::new(numbers);

    for _ in 0..10 {
        let child_numbers = shared_numbers.clone();

        thread::spawn(move || {
            let local_numbers = &child_numbers[..];

            // Work with the local numbers
        });
    }
}
```

Arc Weak

与 `Rc` 类似，`Arc` 也有一个对应的 `Weak` 类型，从 `std::sync::Weak` 引入。

意义与用法与 `Rc Weak` 基本一致，不同的点是这是多线程的版本。故不再赘述。

一个例子

下面这个例子，表述的是如何实现多个对象同时引用另外一个对象。

```
use std::rc::Rc;

struct Owner {
    name: String
}

struct Gadget {
    id: i32,
    owner: Rc<Owner>
```

```
}

fn main() {
    // Create a reference counted Owner.
    let gadget_owner : Rc<Owner> = Rc::new(
        Owner { name: String::from("Gadget Man") }
    );

    // Create Gadgets belonging to gadget_owner. To increment th
e reference
    // count we clone the `Rc<T>` object.
    let gadget1 = Gadget { id: 1, owner: gadget_owner.clone() };
    let gadget2 = Gadget { id: 2, owner: gadget_owner.clone() };

    drop(gadget_owner);

    // Despite dropping gadget_owner, we're still able to print
out the name
    // of the Owner of the Gadgets. This is because we've only d
ropped the
    // reference count object, not the Owner it wraps. As long a
s there are
    // other `Rc<T>` objects pointing at the same Owner, it will
 remain
    // allocated. Notice that the `Rc<T>` wrapper around Gadget.
owner gets
    // automatically dereferenced for us.
    println!("Gadget {} owned by {}", gadget1.id, gadget1.owner.
name);
    println!("Gadget {} owned by {}", gadget2.id, gadget2.owner.
name);

    // At the end of the method, gadget1 and gadget2 get destroy
ed, and with
    // them the last counted references to our Owner. Gadget Man
 now gets
    // destroyed as well.
}
```

Mutex 与 RwLock

Mutex

`Mutex` 意为互斥对象，用来保护共享数据。Mutex 有下面几个特征：

1. `Mutex` 会等待获取锁令牌(token)，在等待过程中，会阻塞线程。直到锁令牌得到。同时只有一个线程的 `Mutex` 对象获取到锁；
2. `Mutex` 通过 `.lock()` 或 `.try_lock()` 来尝试得到锁令牌，被保护的对象，必须通过这两个方法返回的 `RAII` 守卫来调用，不能直接操作；
3. 当 `RAII` 守卫作用域结束后，锁会自动解开；
4. 在多线程中，`Mutex` 一般和 `Arc` 配合使用。

示例：

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc::channel;

const N: usize = 10;

// Spawn a few threads to increment a shared variable (non-atomi
cally), and
// let the main thread know once all increments are done.
//
// Here we're using an Arc to share memory among threads, and th
e data inside
// the Arc is protected with a mutex.
let data = Arc::new(Mutex::new(0));

let (tx, rx) = channel();
for _ in 0..10 {
    let (data, tx) = (data.clone(), tx.clone());
    thread::spawn(move || {
        // The shared state can only be accessed once the lock i
s held.
        // Our non-atomic increment is safe because we're the on
```

```
ly thread
        // which can access the shared state when the lock is he
ld.
        //
        // We unwrap() the return value to assert that we are no
t expecting
        // threads to ever fail while holding the lock.
        let mut data = data.lock().unwrap();
        *data += 1;
        if *data == N {
            tx.send(()).unwrap();
        }
        // the lock is unlocked here when `data` goes out of sco
pe.
    });
}

rx.recv().unwrap();
```

`lock` 与 `try_lock` 的区别

`.lock()` 方法，会等待锁令牌，等待的时候，会阻塞当前线程。而 `.try_lock()` 方法，只是做一次尝试操作，不会阻塞当前线程。

当 `.try_lock()` 没有获取到锁令牌时，会返回 `Err` 。因此，如果要使用 `.try_lock()` ，需要对返回值做仔细处理（比如，在一个循环检查中）。

点评：Rust 的 Mutex 设计成一个对象，不同于 C 语言中的自旋锁用两条分开的语句的实现，更安全，更美观，也更好管理。

RwLock

`RwLock` 翻译成 `读写锁` 。它的特点是：

1. 同时允许多个读，最多只能有一个写；
2. 读和写不能同时存在；

比如：

```
use std::sync::RwLock;

let lock = RwLock::new(5);

// many reader locks can be held at once
{
    let r1 = lock.read().unwrap();
    let r2 = lock.read().unwrap();
    assert_eq!(*r1, 5);
    assert_eq!(*r2, 5);
} // read locks are dropped at this point

// only one write lock may be held, however
{
    let mut w = lock.write().unwrap();
    *w += 1;
    assert_eq!(*w, 6);
} // write lock is dropped here
```

读写锁的方法

1. `.read()`
2. `.try_read()`
3. `.write()`
4. `.try_write()`

注意需要对 `.try_read()` 和 `.try_write()` 的返回值进行判断。

Cell, RefCell

前面我们提到，Rust 通过其所有权机制，严格控制拥有和借用关系，来保证程序的安全，并且这种安全是在编译期可计算、可预测的。但是这种严格的控制，有时也会带来灵活性的丧失，有的场景下甚至还满足不了需求。

因此，Rust 标准库中，设计了这样一个系统的组件：`Cell`, `RefCell`，它们弥补了 Rust 所有权机制在灵活性上和某些场景下的不足。同时，又没有打破 Rust 的核心设计。它们的出现，使得 Rust 革命性的语言理论设计更加完整，更加实用。

具体是因为，它们提供了 `内部可变性`（相对于标准的 `继承可变性` 来讲的）。

通常，我们要修改一个对象，必须

1. 成为它的拥有者，并且声明 `mut`；
2. 或 以 `&mut` 的形式，借用；

而通过 `Cell`, `RefCell`，我们可以在需要的时候，就可以修改里面的对象。而不受编译期静态借用规则束缚。

`Cell`

`Cell` 有如下特点：

1. `Cell<T>` 只能用于 `T` 实现了 `Copy` 的情况；

`.get()`

`.get()` 方法，返回内部值的一个拷贝。比如：

```
use std::cell::Cell;

let c = Cell::new(5);

let five = c.get();
```

`.set()`

`.set()` 方法，更新值。

```
use std::cell::Cell;

let c = Cell::new(5);

c.set(10);
```

`RefCell`

相对于 `Cell` 只能包裹实现了 `Copy` 的类型，`RefCell` 用于更普遍的情况（其它情况都用 `RefCell` ）。

相对于标准情况的 `静态借用` ，`RefCell` 实现了 `运行时借用` ，这个借用是临时的。这意味着，编译器对 `RefCell` 中的内容，不会做静态借用检查，也意味着，出了什么问题，用户自己负责。

`RefCell` 的特点：

1. 在不确定一个对象是否实现了 `Copy` 时，直接选 `RefCell` ；
2. 如果被包裹对象，同时被可变借用了两次，则会导致线程崩溃。所以需要用户自行判断；
3. `RefCell` 只能用于线程内部，不能跨线程；
4. `RefCell` 常常与 `Rc` 配合使用（都是单线程内部使用）；

我们来看实例：

```
use std::collections::HashMap;
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let shared_map: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMa
p::new()));
    shared_map.borrow_mut().insert("africa", 92388);
    shared_map.borrow_mut().insert("kyoto", 11837);
    shared_map.borrow_mut().insert("piccadilly", 11826);
    shared_map.borrow_mut().insert("marbles", 38);
}
```

从上例可看出，用了 `RefCell` 后，外面是 `不可变引用` 的情况，一样地可以修改被包裹的对象。

常用方法

`.borrow()`

不可变借用被包裹值。同时可存在多个不可变借用。

比如：

```
use std::cell::RefCell;

let c = RefCell::new(5);

let borrowed_five = c.borrow();
let borrowed_five2 = c.borrow();
```

下面的例子会崩溃：

```
use std::cell::RefCell;
use std::thread;

let result = thread::spawn(move || {
   let c = RefCell::new(5);
   let m = c.borrow_mut();

   let b = c.borrow(); // this causes a panic
}).join();

assert!(result.is_err());
```

`.borrow_mut()`

可变借用被包裹值。同时只能有一个可变借用。

比如：

```
use std::cell::RefCell;

let c = RefCell::new(5);

let borrowed_five = c.borrow_mut();
```

下面的例子会崩溃：

```
use std::cell::RefCell;
use std::thread;

let result = thread::spawn(move || {
   let c = RefCell::new(5);
   let m = c.borrow();

   let b = c.borrow_mut(); // this causes a panic
}).join();

assert!(result.is_err());
```

`.into_inner()`

取出包裹值。

```
use std::cell::RefCell;

let c = RefCell::new(5);

let five = c.into_inner();
```

一个综合示例

下面这个示例，表述的是如何实现两个对象的循环引用。综合演示了 `Rc` , `Weak` , `RefCell` 的用法

```
use std::rc::Rc;
use std::rc::Weak;
use std::cell::RefCell;

struct Owner {
    name: String,
    gadgets: RefCell<Vec<Weak<Gadget>>>,
    // 其他字段
}

struct Gadget {
    id: i32,
    owner: Rc<Owner>,
    // 其他字段
}

fn main() {
    // 创建一个可计数的Owner。
    // 注意我们将gadgets赋给了Owner。
    // 也就是在这个结构体里， gadget_owner包含gadets
    let gadget_owner : Rc<Owner> = Rc::new(
        Owner {
```

```
            name: "Gadget Man".to_string(),
            gadgets: RefCell::new(Vec::new()),
        }
    );

    // 首先，我们创建两个gadget，他们分别持有 gadget_owner 的一个引用。
    let gadget1 = Rc::new(Gadget{id: 1, owner: gadget_owner.clon
e()});
    let gadget2 = Rc::new(Gadget{id: 2, owner: gadget_owner.clon
e()});

    // 我们将从gadget_owner的gadgets字段中持有其可变引用
    // 然后将两个gadget的Weak引用传给owner。
    gadget_owner.gadgets.borrow_mut().push(Rc::downgrade(&gadget
1));
    gadget_owner.gadgets.borrow_mut().push(Rc::downgrade(&gadget
2));

    // 遍历 gadget_owner的gadgets字段
    for gadget_opt in gadget_owner.gadgets.borrow().iter() {

        // gadget_opt 是一个 Weak<Gadget> 。 因为 weak 指针不能保证
他所引用的对象
        // 仍然存在。所以我们需要显式的调用 upgrade() 来通过其返回值(Opt
ion<_>)来判
        // 断其所指向的对象是否存在。
        // 当然，这个Option为None的时候这个引用原对象就不存在了。
        let gadget = gadget_opt.upgrade().unwrap();
        println!("Gadget {} owned by {}", gadget.id, gadget.owne
r.name);
    }

    // 在main函数的最后， gadget_owner, gadget1和daget2都被销毁。
    // 具体是，因为这几个结构体之间没有了强引用（`Rc<T>`），所以，当他们销
毁的时候。
    // 首先 gadget1和gadget2被销毁。
    // 然后因为gadget_owner的引用数量为0，所以这个对象可以被销毁了。
    // 循环引用问题也就避免了
}
```

类型系统中的几个常见 trait

本章讲解 Rust 类型系统中的几个常见 trait。有 `Into, From, AsRef, AsMut, Borrow, BorrowMut, ToOwned, Deref, Cow` 。

Into/From 及其在 String 和 &str 互转上的应用

`std::convert` 下面，有两个 Trait，`Into/From`，它们是一对孪生姐妹。它们的作用是配合泛型，进行一些设计上的归一化处理。

它们的基本形式为： `From<T>` 和 `Into<T>` 。

From

对于类型为 `U` 的对象 `foo`，如果它实现了 `From<T>`，那么，可以通过 `let foo = U::from(bar)` 来生成自己。这里，`bar` 是类型为 `T` 的对象。

下面举一例，因为 `String` 实现了 `From<&str>`，所以 `String` 可以从 `&str` 生成。

```
let string = "hello".to_string();
let other_string = String::from("hello");

assert_eq!(string, other_string);
```

Into

对于一个类型为 `U: Into<T>` 的对象 `foo`，`Into` 提供了一个函数： `.into(self) -> T`，调用 `foo.into()` 会消耗自己（转移资源所有权），生成类型为 `T` 的另一个新对象 `bar` 。

这句话，说起来有点抽象。下面拿一个具体的实例来辅助理解。

```
fn is_hello<T: Into<Vec<u8>>>(s: T) {
   let bytes = b"hello".to_vec();
   assert_eq!(bytes, s.into());
}

let s = "hello".to_string();
is_hello(s);
```

因为 `String` 类型实现了 `Into<Vec<u8>>` 。

下面拿一个实际生产中字符串作为函数参数的例子来说明。

在我们设计库的 `API` 的时候，经常会遇到一个恼人的问题，函数参数如果定为 `String` ，则外部传入实参的时候，对字符串字面量，必须要做 `.to_string()` 或 `.to_owned()` 转换，参数一多，就是一件又乏味又丑的事情。（而反过来设计的话，对初学者来说，又会遇到一些生命周期的问题，比较麻烦，这个后面论述）

那存不存在一种方法，能够使传参又能够接受 `String` 类型，又能够接受 `&str` 类型呢？答案就是泛型。而仅是泛型的话，太宽泛。因此，标准库中，提供了 `Into<T>` 来为其做约束，以便方便而高效地达到我们的目的。

比如，我们有如下结构体：

```
struct Person {
    name: String,
}

impl Person {
    fn new (name: String) -> Person {
        Person { name: name }
    }
}
```

我们在调用的时候，是这样的：

```
let name = "Herman".to_string();
let person = Person::new(name);
```

如果直接写成：

```
let person = Person::new("Herman");
```

就会报类型不匹配的错误。

好了，下面 `Into` 出场。我们可以定义结构体为

```
struct Person {
    name: String,
}

impl Person {
    fn new<S: Into<String>>(name: S) -> Person {
        Person { name: name.into() }
    }
}
```

然后，调用的时候，下面两种写法都是可以的：

```
fn main() {
    let person = Person::new("Herman");
    let person = Person::new("Herman".to_string());
}
```

我们来仔细分析一下这一块的写法

```
impl Person {
    fn new<S: Into<String>>(name: S) -> Person {
        Person { name: name.into() }
    }
}
```

参数类型为 `S`，是一个泛型参数，表示可以接受不同的类型。`S: Into<String>` 表示 `S` 类型必须实现了 `Into<String>`（约束）。而 `&str` 类型，符合这个要求。因此 `&str` 类型可以直接传进来。

而 `String` 本身也是实现了 `Into<String>` 的。当然也可以直接传进来。

然后，下面 `name: name.into()` 这里也挺神秘的。它的作用是将 `name` 转换成 `String` 类型的另一个对象。当 name 是 `&str` 时，它会转换成 `String` 对象，会做一次字符串的拷贝（内存的申请、复制）。而当 name 本身是 `String` 类型时，`name.into()` 不会做任何转换，代价为零（有没有恍然大悟）。

根据参考资料，上述内容通过下面三式获得：

```
impl<'a> From<&'a str> for String {}
impl<T> From<T> for T {}
impl<T, U> Into<U> for T where U: From<T> {}
```

更多内容，请参考：

- http://doc.rust-lang.org/std/convert/trait.Into.html
- http://doc.rust-lang.org/std/convert/trait.From.html
- http://hermanradtke.com/2015/05/06/creating-a-rust-function-that-accepts-string-or-str.html

AsRef 和 AsMut

`std::convert` 下面，还有另外两个 Trait， `AsRef/AsMut` ，它们功能是配合泛型，在执行引用操作的时候，进行自动类型转换。这能够使一些场景的代码实现得清晰漂亮，大家方便开发。

AsRef

`AsRef` 提供了一个方法 `.as_ref()` 。

对于一个类型为 `T` 的对象 `foo` ，如果 `T` 实现了 `AsRef<U>` ，那么， `foo` 可执行 `.as_ref()` 操作，即 `foo.as_ref()` 。操作的结果，我们得到了一个类型为 `&U` 的新引用。

注：

1. 与 `Into<T>` 不同的是， `AsRef<T>` 只是类型转换， `foo` 对象本身没有被消耗；
2. `T: AsRef<U>` 中的 `T` ，可以接受 资源拥有者（owned）类型，共享引用（shared referrence）类型 ，可变引用（mutable referrence）类型。

下面举个简单的例子：

```
fn is_hello<T: AsRef<str>>(s: T) {
   assert_eq!("hello", s.as_ref());
}

let s = "hello";
is_hello(s);

let s = "hello".to_string();
is_hello(s);
```

因为 `String` 和 `&str` 都实现了 `AsRef<str>` 。

AsMut

`AsMut<T>` 提供了一个方法 `.as_mut()` 。它是 `AsRef<T>` 的可变（mutable）引用版本。

对于一个类型为 `T` 的对象 `foo` ，如果 `T` 实现了 `AsMut<U>` ，那么， `foo` 可执行 `.as_mut()` 操作，即 `foo.as_mut()` 。操作的结果，我们得到了一个类型为 `&mut U` 的可变（mutable）引用。

注：在转换的过程中， `foo` 会被可变（mutable）借用。

Borrow, BorrowMut, ToOwned

Borrow

`use std::borrow::Borrow;`

`Borrow` 提供了一个方法 `.borrow()` 。

对于一个类型为 `T` 的值 `foo` ，如果 `T` 实现了 `Borrow<U>` ，那么， `foo` 可执行 `.borrow()` 操作，即 `foo.borrow()` 。操作的结果，我们得到了一个类型为 `&U` 的新引用。

`Borrow` 可以认为是 `AsRef` 的严格版本，它对普适引用操作的前后类型之间附加了一些其它限制。

`Borrow` 的前后类型之间要求必须有内部等价性。不具有这个等价性的两个类型之间，不能实现 `Borrow` 。

`AsRef` 更通用，更普遍，覆盖类型更多，是 `Borrow` 的超集。

举例：

```
use std::borrow::Borrow;

fn check<T: Borrow<str>>(s: T) {
    assert_eq!("Hello", s.borrow());
}

let s = "Hello".to_string();

check(s);

let s = "Hello";

check(s);
```

BorrowMut

```
use std::borrow::BorrowMut;
```

`BorrowMut<T>` 提供了一个方法 `.borrow_mut()` 。它是 `Borrow<T>` 的可变（mutable）引用版本。

对于一个类型为 `T` 的值 `foo` ，如果 `T` 实现了 `BorrowMut<U>` ，那么， `foo` 可执行 `.borrow_mut()` 操作，即 `foo.borrow_mut()` 。操作的结果我们得到类型为 `&mut U` 的一个可变（mutable）引用。

注：在转换的过程中， `foo` 会被可变（mutable）借用。

ToOwned

```
use std::borrow::ToOwned;
```

`ToOwned` 为 `Clone` 的普适版本。它提供了 `.to_owned()` 方法，用于类型转换。

有些实现了 `Clone` 的类型 `T` 可以从引用状态实例 `&T` 通过 `.clone()` 方法，生成具有所有权的 `T` 的实例。但是它只能由 `&T` 生成 `T` 。而对于其它形式的引用， `Clone` 就无能为力了。

而 `ToOwned` trait 能够从任意引用类型实例，生成具有所有权的类型实例。

参考

- http://doc.rust-lang.org/std/borrow/trait.Borrow.html

Deref

`Deref` 是 `deref` 操作符 `*` 的 trait，比如 `*v` 。

一般理解， `*v` 操作，是 `&v` 的反向操作，即试图由资源的引用获取到资源的拷贝（如果资源类型实现了 `Copy` ），或所有权（资源类型没有实现 `Copy` ）。

Rust 中，本操作符行为可以重载。这也是 Rust 操作符的基本特点。本身没有什么特别的。

强制隐式转换（coercion）

`Deref` 神奇的地方并不在本身 `解引` 这个意义上，Rust 的设计者在它之上附加了一个特性： `强制隐式转换` ，这才是它神奇之处。

这种隐式转换的规则为：

一个类型为 `T` 的对象 `foo` ，如果 `T: Deref<Target=U>` ，那么，相关 `foo` 的某个智能指针或引用（比如 `&foo` ）在应用的时候会自动转换成 `&U` 。

粗看这条规则，貌似有点类似于 `AsRef` ，而跟 `解引` 似乎风马牛不相及。实际里面有些玄妙之处。

Rust 编译器会在做 `*v` 操作的时候，自动先把 `v` 做引用归一化操作，即转换成内部通用引用的形式 `&v` ，整个表达式就变成 `*&v` 。这里面有两种情况：

1. 把其它类型的指针（比如在库中定义的， `Box` , `Rc` , `Arc` , `Cow` 等），转成内部标准形式 `&v` ；
2. 把多重 `&` （比如： `&&&&&&&v` ），简化成 `&v` （通过插入足够数量的 `*` 进行解引）。

所以，它实际上在解引用之前做了一个引用的归一化操作。

为什么要转呢？ 因为编译器设计的能力是，只能够对 `&v` 这种引用进行解引用。其它形式的它不认识，所以要做引用归一化操作。

使用引用进行过渡也是为了能够防止不必要的拷贝。

下面举一些例子：

```
fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();

// therefore, this works:
foo(&owned);
```

因为 `String` 实现了 `Deref<Target=str>` 。

```
use std::rc::Rc;

fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// therefore, this works:
foo(&counted);
```

因为 `Rc<T>` 实现了 `Deref<Target=T>` 。

```
fn foo(s: &[i32]) {
    // borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

因为 `Vec<T>` 实现了 `Deref<Target=[T]>` 。

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

上面那几种函数的调用，效果是一样的。

`coercion` 的设计，是 Rust 中仅有的类型隐式转换，设计它的目的，是为了简化程序的书写，让代码不至于过于繁琐。把人从无尽的类型细节中解脱出来，让书写 Rust 代码变成一件快乐的事情。

Cow

直译为奶牛！开玩笑。 `Cow` 是一个枚举类型，通过 `use std::borrow::Cow;` 引入。它的定义是 `Clone-on-write` ，即写时克隆。本质上是一个智能指针。

它有两个可选值：

- `Borrowed` ，用于包裹对象的引用（通用引用）；
- `Owned` ，用于包裹对象的所有者；

`Cow` 提供

1. 对此对象的不可变访问（比如可直接调用此对象原有的不可变方法）；
2. 如果遇到需要修改此对象，或者需要获得此对象的所有权的情况， `Cow` 提供方法做克隆处理，并避免多次重复克隆。

`Cow` 的设计目的是提高性能（减少复制）同时增加灵活性，因为大部分情况下，业务场景都是读多写少。利用 `Cow` ，可以用统一，规范的形式实现，需要写的时候才做一次对象复制。这样就可能会大大减少复制的次数。

它有以下几个要点需要掌握：

1. `Cow<T>` 能直接调用 `T` 的不可变方法，因为 `Cow` 这个枚举，实现了 `Deref` ；
2. 在需要写 `T` 的时候，可以使用 `.to_mut()` 方法得到一个具有所有权的值的可变借用；
    i. 注意，调用 `.to_mut()` 不一定会产生克隆；
    ii. 在已经具有所有权的情况下，调用 `.to_mut()` 有效，但是不会产生新的克隆；
    iii. 多次调用 `.to_mut()` 只会产生一次克隆。
3. 在需要写 `T` 的时候，可以使用 `.into_owned()` 创建新的拥有所有权的对象，这个过程往往意味着内存拷贝并创建新对象；
    i. 如果之前 `Cow` 中的值是借用状态，调用此操作将执行克隆；
    ii. 本方法，参数是 `self` 类型，它会“吃掉”原先的那个对象，调用之后原先的对象的生命周期就截止了，在 `Cow` 上不能调用多次；

举例

`.to_mut()` 举例

```
use std::borrow::Cow;

let mut cow: Cow<[_]> = Cow::Owned(vec![1, 2, 3]);

let hello = cow.to_mut();

assert_eq!(hello, &[1, 2, 3]);
```

`.into_owned()` 举例

```
use std::borrow::Cow;

let cow: Cow<[_]> = Cow::Owned(vec![1, 2, 3]);

let hello = cow.into_owned();

assert_eq!(vec![1, 2, 3], hello);
```

综合举例

```
use std::borrow::Cow;

fn abs_all(input: &mut Cow<[i32]>) {
    for i in 0..input.len() {
        let v = input[i];
        if v < 0 {
            // clones into a vector the first time (if not alrea
dy owned)
            input.to_mut()[i] = -v;
        }
    }
}
```

`Cow` 在函数返回值上的应用实例

题目：写一个函数，过滤掉输入的字符串中的所有空格字符，并返回过滤后的字符串。

对这个简单的问题，不用思考，我们都可以很快写出代码：

```
fn remove_spaces(input: &str) -> String {
    let mut buf = String::with_capacity(input.len());

    for c in input.chars() {
        if c != ' ' {
            buf.push(c);
        }
    }

    buf
}
```

设计函数输入参数的时候，我们会停顿一下，这里，用 `&str` 好呢，还是 `String` 好呢？思考一番，从性能上考虑，有如下结论：

1. 如果使用 `String` ，则外部在调用此函数的时候，
    i. 如果外部的字符串是 `&str` ，那么，它需要做一次克隆，才能调用此函数；
    ii. 如果外部的字符串是 `String` ，那么，它不需要做克隆，就可以调用此函数。但是，一旦调用后，外部那个字符串的所有权就被 `move` 到此函数中了，外部的后续代码将无法再使用原字符串。
2. 如果使用 `&str` ，则不存在上述两个问题。但可能会遇到生命周期的问题，需要注意。

继续分析上面的例子，我们发现，在函数体内，做了一次新字符串对象的生成和拷贝。

让我们来仔细分析一下业务需求。最坏的情况下，如果字符串中没有空白字符，那最好是直接原样返回。这种情况做这样一次对象的拷贝，完全就是浪费了。

于是我们心想改进这个算法。很快，又遇到了另一个问题，返回值是 `String` 的嘛，我不论怎样，要把 `&str` 转换成 `String` 返回，始终都要经历一次复制。于是我们快要放弃了。

好吧， `Cow` 君这时出马了。奶牛君很快写出了如下代码：

```
use std::borrow::Cow;

fn remove_spaces<'a>(input: &'a str) -> Cow<'a, str> {
    if input.contains(' ') {
        let mut buf = String::with_capacity(input.len());

        for c in input.chars() {
            if c != ' ' {
                buf.push(c);
            }
        }

        return Cow::Owned(buf);
    }

    return Cow::Borrowed(input);
}
```

完美解决了业务逻辑与返回值类型冲突的问题。本例可细细品味。

外部程序，拿到这个 `Cow` 返回值后，按照我们上文描述的 `Cow` 的特性使用就好了。

Send 和 Sync

`std::marker` 模块中，有两个 trait：`Send` 和 `Sync`，它们与多线程安全相关。

标记为 `marker trait` 的 trait，它实际就是一种约定，没有方法的定义，也没有关联元素（associated items）。仅仅是一种约定，实现了它的类型必须满足这种约定。一种类型是否加上这种约定，要么是编译器的行为，要么是人工手动的行为。

`Send` 和 `Sync` 在大部分情况下（针对 Rust 的基础类型和 std 中的大部分类型），会由编译器自动推导出来。对于不能由编译器自动推导出来的类型，要使它们具有 `Send` 或 `Sync` 的约定，可以由人手动实现。实现的时候，必须使用 `unsafe` 前缀，因为 Rust 默认不信任程序员，由程序员自己控制的东西，统统标记为 `unsafe`，出了问题（比如，把不是线程安全的对象加上 `Sync` 约定）由程序员自行负责。

它们的定义如下：

如果 `T: Send`，那么将 `T` 传到另一个线程中时（按值传送），不会导致数据竞争或其它不安全情况。

1. `Send` 是对象可以安全发送到另一个执行体中；
2. `Send` 使被发送对象可以和产生它的线程解耦，防止原线程将此资源释放后，在目标线程中使用出错（use after free）。

如果 `T: Sync`，那么将 `&T` 传到另一个线程中时，不会导致数据竞争或其它不安全情况。

1. `Sync` 是可以被同时多个执行体访问而不出错；
2. `Sync` 防止的是竞争；

推论：

1. `T: Sync` 意味着 `&T: Send`；
2. `Sync + Copy = Send`；
3. 当 `T: Send` 时，可推导出 `&mut T: Send`；
4. 当 `T: Sync` 时，可推导出 `&mut T: Sync`；

5. 当 `&mut T: Send` 时，不能推导出 `T: Send` ；

（注： `T` , `&T` , `&mut T` ， `Box<T>` 等都是不同的类型）

具体的类型：

1. 原始类型（比如： u8, f64），都是 `Sync` ，都是 `Copy` ，因此都是 `Send` ；
2. 只包含原始类型的复合类型，都是 `Sync` ，都是 `Copy` ，因此都是 `Send` ；
3. 当 `T: Sync` ， `Box<T>` , `Vec<T>` 等集合类型是 `Sync` ；
4. 具有内部可变性的的指针，不是 `Sync` 的，比如 `Cell` , `RefCell` , `UnsafeCell` ；
5. `Rc` 不是 `Sync` 。因为只要一做 `&Rc<T>` 操作，就会克隆一个新引用，它会以非原子性的方式修改引用计数，所以是不安全的；
6. 被 `Mutex` 和 `RWLock` 锁住的类型 `T: Send` ，是 `Sync` 的；
7. 原始指针（ `*mut` , `*const` ）既不是 `Send` 也不是 `Sync` ；

Rust 正是通过这两大武器： `所有权和生命周期` + `Send 和 Sync` （本质上为类型系统）来为并发编程提供了安全可靠的基础设施。使得程序员可以放心在其上构建稳健的并发模型。这也正是 Rust 的核心设计观的体现：内核只提供最基础的原语，真正的实现能分离出去就分离出去。并发也是如此。

并发，并行，多线程编程

本章讲解 Rust 中，并发，并行，多线程编程的相关知识。

并发编程

并发是什么？引用Rob Pike的经典描述:

> 并发是同一时间应对多件事情的能力

其实在我们身边就有很多并发的事情，比如一边上课，一边发短信；一边给小孩喂奶，一边看电视，只要你细心留意，就会发现许多类似的事。相应地，在软件的世界里，我们也会发现这样的事，比如一边写博客，一边听音乐；一边看网页，一边下载软件等等。显而易见这样会节约不少时间，干更多的事。然而一开始计算机系统并不能同时处理两件事，这明显满足不了我们的需要，后来慢慢提出了多进程，多线程的解决方案，再后来，硬件也发展到了多核多CPU的地步。在硬件和系统底层对并发的支持也来越多，相应地，各大编程语言也对并发处理提供了强力的支持，作为新兴语言的Rust，自然也支持并发编程。那么本章就将引领大家一览Rust并发编程的相关知识，从线程开始，逐步尝试进行数据交互，同步协作，最后进入到并行实现，一步一步揭开Rust并发编程的神秘面纱。由于本书主要介绍的是Rust语言的使用，所以本章不会对并发编程相关理论知识进行全面而深入地探讨——要眞那样地话，一本书都不够介绍的，而是更侧重于介绍用Rust语言怎么实现基本的并发。

首先我们会介绍线程的使用，线程是基本的执行单元，其重要性不言而喻，Rust程序就是由一堆线程组成的。在当今多核多CPU已经普及的情况下，各种大数据分析和并行计算又让线程焕发出了更耀眼的光芒。如果对线程不甚了解，请先参阅操作系统相关的书籍，此处不过多介绍。然后介绍一些在解决并发问题时，需要处理的数据传递和协作的实现，比如消息传递，同步和共享内存。最后简要介绍Rust中并行的实现。

24.1 线程创建与结束

相信线程对大家而言，一点也不陌生，在当今多CPU多核已经普及的情况下，大数据分析与并行计算都离不开它，几乎所有的语言都支持它，所有的进程都是由一个或多个线程所组成的。既然如此重要，接下来我们就先来看一下在Rust中如何创建一个线程，然后线程又是如何结束的。

Rust对于线程的支持，和 `C++11` 一样，都是放在标准库中来实现的，详情请参见 `std::thread` ，好在Rust从一开始就这样做了，不用像C++那样等呀等。在语言层面支持后，开发者就不用那么苦兮兮地处理各平台的移植问题。通过Rust的源码可以看到， `std::thread` 其实就是对不同平台的线程操作的封装，相关API的实现都是调用操作系统的API来实现的，从而提供了线程操作的统一接口。对于我而言，能够这样简单快捷地操作原生线程，身上的压力一下轻了不少。

创建线程

首先，我们看一下在Rust中如何创建一个原生线程(native thread)。 `std::thread` 提供了两种创建方式，都非常简单，第一种方式是通过 `spawn` 函数来创建，参见下面的示例代码：

```
use std::thread;

fn main() {
    // 创建一个线程
    let new_thread = thread::spawn(move || {
        println!("I am a new thread.");
    });
    // 等待新建线程执行完成
    new_thread.join().unwrap();
}
```

执行上面这段代码，将会看到下面的输出结果：

```
I am a new thread.
```

就5行代码，少得不能再少，最关键的当然就是调用 `spawn` 函数的那行代码。使用这个函数，记得要先 `use std::thread` 。注意 `spawn` 函数需要一个函数作为参数，且是 `FnOnce` 类型，如果已经忘了这种类型的函数，请学习或回顾一下函数和闭包章节。 `main` 函数最后一行代码即使不要，也能创建线程（关于 `join` 函数的作用和使用在后续小节详解，此处你只要知道它可以用来等待线程执行完成即可），可以去掉或者注释该行代码试试。这样的话，运行结果可能没有任何输出，具体原因后面详解。

接下来我们使用第二种方式创建线程，它比第一种方式稍微复杂一点，因为功能强大一点，可以在创建之前设置线程的名称和堆栈大小，参见下面的代码：

```
use std::thread;

fn main() {
    // 创建一个线程，线程名称为 thread1, 堆栈大小为4k
    let new_thread_result = thread::Builder::new()
                            .name("thread1".to_string())
                            .stack_size(4*1024*1024).spawn(move
|| {
        println!("I am thread1.");
    });
    // 等待新创建的线程执行完成
    new_thread_result.unwrap().join().unwrap();
}
```

执行上面这段代码，将会看到下面的输出结果：

```
I am thread1.
```

通过和第一种方式的实现代码比较可以发现，这种方式借助了一个 `Builder` 类来设置线程名称和堆栈大小，除此之外， `Builder` 的 `spawn` 函数的返回值是一个 `Result` ，在正式的代码编写中，可不能像上面这样直接 `unwrap.join` ，应该判定一下。后面也会有很多类似的演示代码，为了简单说明不会做的很严谨。

以上就是Rust创建原生线程的两种不同方式，示例代码有点然并卵的意味，但是你可以稍加修改，就可以变得更加有用，试试吧。

线程结束

此时，我们已经知道如何创建一个新线程了，创建后，不管你见或者不见，它就在那里，那么它什么时候才会消亡呢？自生自灭，亦或者被干掉？如果接触过一些系统编程，应该知道有些操作系统提供了粗暴地干掉线程的接口，看它不爽，直接干掉，完全可以不理会新建线程的感受。是否感觉很爽，但是Rust不会再让这样爽了，因为 `std::thread` 并没有提供这样的接口，为什么呢？如果深入接触过并发编程或多线程编程，就会知道强制终止一个运行中的线程，会出现诸多问题。比如

资源没有释放，引起状态混乱，结果不可预期。强制干掉那一刻，貌似很爽地解决问题了，然而可能后患无穷。Rust语言的一大特性就是安全，是绝对不允许这样不负责任的做法的。即使在其他语言提供了类似的接口，也不应该滥用。

那么在Rust中，新建的线程就只能让它自身自灭了吗？其实也有两种方式，首先介绍大家都知道的自生自灭的方式，线程执行体执行完成，线程就结束了。比如上面创建线程的第一种方式，代码执行完 `println!("I am a new thread.");` 就结束了。 如果像下面这样：

```
use std::thread;

fn main() {
    // 创建一个线程
    let new_thread = thread::spawn(move || {
        loop {
            println!("I am a new thread.");
        }
    });
    // 等待新创建的线程执行完成
    new_thread.join().unwrap();
}
```

线程就永远都不会结束，如果你用的还是古董电脑，运行上面的代码之前，请做好心理准备。在实际代码中，要时刻警惕该情况的出现（单核情况下，CPU占用率会飙升到100%），除非你是故意为之。

线程结束的另一种方式就是，线程所在进程结束了。我们把上面这个例子稍作修改：

```
use std::thread;

fn main() {
    // 创建一个线程
    thread::spawn(move || {
        loop {
            println!("I am a new thread.");
        }
    });

    // 不等待新创建的线程执行完成
    // new_thread.join().unwrap();
}
```

同上面的代码相比，唯一的差别在于 `main` 函数的最后一行代码被注释了，这样主线程就不用等待新建线程了，在创建线程之后就执行完了，其所在进程也就结束了，从而新建的线程也就结束了。此处，你可能有疑问：为什么一定是进程结束导致新建线程结束？也可能是创建新线程的主线程结束而导致的？事实到底如何，我们不妨验证一下：

```
use std::thread;

fn main() {
    // 创建一个线程
    let new_thread = thread::spawn(move || {
        // 再创建一个线程
        thread::spawn(move || {
            loop {
                println!("I am a new thread.");
            }
        })
    });

    // 等待新创建的线程执行完成
    new_thread.join().unwrap();
    println!("Child thread is finish!");

    // 睡眠一段时间，看子线程创建的子线程是否还在运行
    thread::sleep_ms(100);
}
```

这次我们在新建线程中还创建了一个线程，从而第一个新建线程是父线程，主线程在等待该父线程结束后，主动睡眠一段时间。这样做有两个目的，一是确保整个程序不会马上结束；二是如果子线程还存在，应该会获得执行机会，以此来检验子线程是否还在运行，下面是输出结果：

```
Child thread is finish!
I am a new thread.
I am a new thread.
......
```

结果表明，在父线程结束后，其创建的子线程还活着，这并不会因为父线程结束而结束。这个还是比较符合自然规律的，要不然真会断子绝孙，人类灭绝。所以导致线程结束的第二种方式，是结束其所在进程。到此为止，我们已经把线程的创建和结束都介绍完了，那么接下来我们会介绍一些更有趣的东西。但是在此之前，请先考虑一下下面的练习题。

练习题：

有一组学生的成绩，我们需要对它们评分，90分及以上是A，80分及以上是B，70分及以上是C，60分及以上为D,60分以下为E。现在要求用Rust语言编写一个程序来评分，且评分由新建的线程来做，最终输出每个学生的学号，成绩，评分。学生成绩单随机产生，学生人数100位，成绩范围为[0,100]，学号依次从1开始，直到100。

消息传递

稍加考虑，上一节的练习题其实是不完整的，它只是评分系统中的一环，一个评分系统是需要先把信息从数据库或文件中读取出来，然后才是评分，最后还需要把评分结果再保存到数据库或文件中去。如果一步一步串行地做这三个步骤，是完全没有问题的。那么我们是否可以用三个线程来分别做这三个步骤呢？上一节练习题我们已经用了一个线程来实现评分，那么我们是否也可以再用一个线程来读取成绩，再用另个线程来实现保存呢？ 如果能这样的话，那么我们就可以利用上多核多cpu的优势，加快整个评分的效率。既然在此提出这个问题，答案就很明显了。问题在于我们要怎么在Rust中来实现，关键在于三个线程怎么交换信息，以达到串行的逻辑处理顺序？

为了解决这个问题，下面将介绍一种Rust在标准库中支持的消息传递技术。消息传递是并发模型里面大家比较推崇的模式，不仅仅是因为使用起来比较简单，关键还在于它可以减少数据竞争，提高并发效率，为此值得深入学习。Rust是通过一个叫做通道(`channel`)的东西来实现这种模式的，下面直接进入主题。

初试通道(channel)

Rust的通道(`channel`)可以把一个线程的消息(数据)传递到另一个线程，从而让信息在不同的线程中流动，从而实现协作。详情请参见`std::sync::mpsc`。通道的两端分别是发送者(`Sender`)和接收者(`Receiver`)，发送者负责从一个线程发送消息，接收者则在另一个线程中接收该消息。下面我们来看一个简单的例子：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<i32>, mpsc::Receiver<i32>) =
        mpsc::channel();

    // 创建线程用于发送消息
    thread::spawn(move || {
        // 发送一个消息，此处是数字id
        tx.send(1).unwrap();
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}
```

程序说明参见代码中的注释，程序执行结果为：

```
receive 1
```

结果表明 `main` 所在的主线程接收到了新建线程发送的消息，用Rust在线程间传递消息就是这么简单！

虽然简单，但使用过其他语言就会知道，通道有多种使用方式，且比较灵活，为此我们需要进一步考虑关于 `Rust` 的 `Channel` 的几个问题：

1. 通道能保证消息的顺序吗？是否先发送的消息，先接收？
2. 通道能缓存消息吗？如果能的话能缓存多少？
3. 通道的发送者和接收者支持N:1，1:N，N:M模式吗？
4. 通道能发送任何数据吗？
5. 发送后的数据，在线程中继续使用没有问题吗？

让我们带着这些问题和思考进入下一个小节，那里有相关的答案。

消息类型

上面的例子中，我们传递的消息类型为 `i32` ，除了这种类型之外，是否还可以传递更多的原始类型，或者更复杂的类型，和自定义类型？下面我们尝试发送一个更复杂的 `Rc` 类型的消息：

```
use std::fmt;
use std::sync::mpsc;
use std::thread;
use std::rc::Rc;

pub struct Student {
    id: u32
}

impl fmt::Display for Student {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "student {}", self.id)
    }
}

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<Rc<Student>>, mpsc::Receiver<Rc<
Student>>) =
        mpsc::channel();

    // 创建线程用于发送消息
    thread::spawn(move || {
        // 发送一个消息，此处是数字id
        tx.send(Rc::new(Student{
            id: 1,
        })).unwrap();
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}
```

编译代码，奇迹没有出现，编译时错误，错误提示：

```
error: the trait `core::marker::Send` is not
implemented for the type `alloc::rc::Rc<Student>` [E0277]
note: `alloc::rc::Rc<Student>` cannot be sent between threads sa
fely
```

看来并不是所有类型的消息都可以通过通道发送，消息类型必须实现 `marker trait Send` 。Rust之所以这样强制要求，主要是为了解决并发安全的问题，再一次强调，安全是Rust考虑的重中之重。如果一个类型是 `Send` ，则表明它可以在线程间安全的转移所有权(`ownership`)，当所有权从一个线程转移到另一个线程后，同一时间就只会存在一个线程能访问它，这样就避免了数据竞争，从而做到线程安全。 `ownership` 的强大又一次显示出来了。通过这种做法，在编译时即可要求所有的代码必须满足这一约定，这种方式方法值得借鉴， `trait` 也是非常强大。

看起来问题得到了完美的解决，然而由于 `Send` 本身是一个不安全的 `marker trait` ，并没有实际的 `API` ，所以实现它很简单，但没有强制保障，就只能靠开发者自己约束，否则还是可能引发并发安全问题。对于这一点，也不必太过担心，因为Rust中已经存在的类，都已经实现了 `Send` 或 `!Send` ，我们只要使用就行。 `Send` 是一个默认应用到所有Rust已存在类的trait，所以我们用 `!Send` 显式标明该类没有实现 `Send` 。目前几乎所有的原始类型都是 `Send` ，例如前面例子中发送的 `i32` 。对于开发者而言，我们可能会更关心哪些是非 `Send` ，也就是实现了 `!Send` ，因为这会导致线程不安全。更全面的信息参见 `Send` 官网API。

对于不是 `Send` 的情况（ `!Send` ），大致分为两类：

1. 原始指针，包括 `*mut T` 和 `*const T` ，因为不同线程通过指针都可以访问数据，从而可能引发线程安全问题。
2. `Rc` 和 `Weak` 也不是，因为引用计数会被共享，但是并没有做并发控制。

虽然有这些 `!Send` 的情况，但是逃不过编译器的火眼金睛，只要你错误地使用了消息类型，编译器都会给出类似于上面的错误提示。我们要担心的不是这些，因为错误更容易出现在新创建的自定义类，有下面两点需要注意：

1. 如果自定义类的所有字段都是 `Send` ，那么这个自定义类也是 `Send` 。 反之，如果有一个字段是 `!Send` ，那么这个自定义类也是 `!Send` 。 如果类的字段存在递归包含的情况，按照该原则以此类推来推论类是 `Send` 还是 `!Send` 。

2. 在为一个自定义类实现 `Send` 或者 `!Send` 时，必须确保符合它的约定。

到此，消息类型的相关知识已经介绍完了，说了这么久，也该让大家自己练习一下了：请实现一个自定义类，该类包含一个Rc字段，让这个类变成可以在通道中发送的消息类型。

异步通道(Channel)

在粗略地尝试通道之后，是时候更深入一下了。Rust的标准库其实提供了两种类型的通道：异步通道和同步通道。上面的例子都是使用的异步通道，为此这一小节我们优先进一步介绍异步通道，后续再介绍同步通道。异步通道指的是：不管接收者是否正在接收消息，消息发送者在发送消息时都不会阻塞。为了验证这一点，我们尝试多增加一个线程来发送消息：

```
use std::sync::mpsc;
use std::thread;

// 线程数量
const THREAD_COUNT :i32 = 2;

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<i32>, mpsc::Receiver<i32>) = mps
c::channel();

    // 创建线程用于发送消息
    for id in 0..THREAD_COUNT {
        // 注意Sender是可以clone的，这样就可以支持多个发送者
        let thread_tx = tx.clone();
        thread::spawn(move || {
            // 发送一个消息，此处是数字id
            thread_tx.send(id + 1).unwrap();
            println!("send {}", id + 1);
        });
    }

    thread::sleep_ms(2000);
    println!("wake up");
    // 在主线程中接收子线程发送的消息并输出
    for _ in 0..THREAD_COUNT {
        println!("receive {}", rx.recv().unwrap());
    }
}
```

运行结果:

```
send 1
send 2
wake up
receive 1
receive 2
```

在代码中，我们故意让 `main` 所在的主线程睡眠2秒，从而让发送者所在线程优先执行，通过结果可以发现，发送者发送消息时确实没有阻塞。还记得在前面提到过很多关于通道的问题吗？从这个例子里面还发现什么没？除了不阻塞之外，我们还能发现另外的三个特征：

1.通道是可以同时支持多个发送者的，通过 `clone` 的方式来实现。 这类似于 `Rc` 的共享机制。 其实从 `Channel` 所在的库名 `std::sync::mpsc` 也可以知道这点。 因为 `mpsc` 就是多生产者单消费者(Multiple Producers Single Consumer)的简写。 可以有多个发送者,但只能有一个接收者，即支持的N:1模式。

2.异步通道具备消息缓存的功能，因为1和2是在没有接收之前就发了的，在此之后还能接收到这两个消息。

那么通道到底能缓存多少消息？在理论上是无穷的，尝试一下便知：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个通道
    let (tx, rx): (mpsc::Sender<i32>, mpsc::Receiver<i32>) = mps
c::channel();

    // 创建线程用于发送消息
    let new_thread = thread::spawn(move || {
        // 发送无穷多个消息
        let mut i = 0;
        loop {
            i = i + 1;
            // add code here
            println!("send {}", i);
            match tx.send(i) {
                Ok(_) => (),
                Err(e) => {
                    println!("send error: {}, count: {}", e, i);
                    return;
                },
            }
        }
    });

    // 在主线程中接收子线程发送的消息并输出
    new_thread.join().unwrap();
    println!("receive {}", rx.recv().unwrap());
}
```

最后的结果就是耗费内存为止。

3.消息发送和接收的顺序是一致的，满足先进先出原则。

上面介绍的内容大多是关于发送者和通道的，下面开始考察一下接收端。通过上面的几个例子，细心一点的可能已经发现接收者的 `recv` 方法应该会阻塞当前线程，如果不阻塞，在多线程的情况下，发送的消息就不可能接收完全。所以没有发送者

发送消息，那么接收者将会一直等待，这一点要谨记。在某些场景下，一直等待是符合实际需求的。但某些情况下并不需一直等待，那么就可以考虑释放通道，只要通道释放了， `recv` 方法就会立即返回。

异步通道的具有良好的灵活性和扩展性，针对业务需要，可以灵活地应用于实际项目中，实在是必备良药！

同步通道

同步通道在使用上同异步通道一样，接收端也是一样的，唯一的区别在于发送端，我们先来看下面的例子：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个同步通道
    let (tx, rx): (mpsc::SyncSender<i32>, mpsc::Receiver<i32>) =
 mpsc::sync_channel(0);

    // 创建线程用于发送消息
    let new_thread = thread::spawn(move || {
        // 发送一个消息，此处是数字id
        println!("before send");
        tx.send(1).unwrap();
        println!("after send");
    });

    println!("before sleep");
    thread::sleep_ms(5000);
    println!("after sleep");
    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
    new_thread.join().unwrap();
}
```

运行结果：

```
before sleep
before send
after sleep
receive 1
after send
```

除了多了一些输出代码之外，上面这段代码几乎和前面的异步通道的没有什么区别，唯一不同的在于创建同步通道的那行代码。同步通道是 `sync_channel` ，对应的发送者也变成了 `SyncSender` 。为了显示出同步通道的区别，故意添加了一些打印。和异步通道相比，存在两点不同：

1. 同步通道是需要指定缓存的消息个数的，但需要注意的是，最小可以是0，表示没有缓存。
2. 发送者是会被阻塞的。当通道的缓存队列不能再缓存消息时，发送者发送消息时，就会被阻塞。

对照上面两点和运行结果来分析，由于主线程在接收消息前先睡眠了，从而子线程这个时候会被调度执行发送消息，由于通道能缓存的消息为0，而这个时候接收者还没有接收，所以 `tx.send(1).unwrap()` 就会阻塞子线程，直到主线程接收消息，即执行 `println!("receive {}", rx.recv().unwrap());` 。运行结果印证了这点，要是没阻塞，那么在 `before send` 之后就应该是 `after send` 了。

相比较而言，异步通道更没有责任感一些，因为消息发送者一股脑的只管发送，不管接收者是否能快速处理。这样就可能出现通道里面缓存大量的消息得不到处理，从而占用大量的内存，最终导致内存耗尽。而同步通道则能避免这种问题，把接受者的压力能传递到发送者，从而一直传递下去。

共享内存

在消息传递之外，还存在一种广为人知的并发模型，那就是共享内存。其实如果不能共享内存，消息传递也是不能在不同的线程间传递消息，也谈不上在不同的线程间等待和通知了。共享内存是这一切得以发生的基础。如果查看源码，你会发现消息传递的内部实现就是借用了共享内存机制。相对于消息传递而言，共享内存会有更多的竞争，但是不用进行多次拷贝，在某些情况下，也需要考虑使用这种方式来处理。在Rust中，能共享内存的情况，主要体现在下面两个方面：

static

Rust语言中也存在static变量，其生命周期是整个应用程序，并且在内存中某个固定地址处只存在一份实例。所有线程都能够访问到它。这种方式也是最简单和直接的共享方式。几乎大多数语言都存在这种机制。下面简单看一下Rust中多个线程访问static变量的用法：

```
use std::thread;

static VAR: i32 = 5;

fn main() {
    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        println!("static value in new thread: {}", VAR);
    });

    // 等待新线程先运行
    new_thread.join().unwrap();
    println!("static value in main thread: {}", VAR);
}
```

运行结果：

```
static value in new thread: 5
static value in main thread: 5
```

`VAR` 这个 `static` 变量在各线程中可以直接使用，非常方便。当然上面只是读取，那么要修改也是很简单的：

```
use std::thread;

static mut VAR: i32 = 5;

fn main() {
    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        unsafe {
            println!("static value in new thread: {}", VAR);
            VAR = VAR + 1;
        }
    });

    // 等待新线程先运行
    new_thread.join().unwrap();
    unsafe {
        println!("static value in main thread: {}", VAR);
    }
}
```

运行结果：

```
static value in new thread: 5
static value in main thread: 6
```

从结果来看 `VAR` 的值变了，从代码上来看，除了在 `VAR` 变量前面加了 `mut` 关键字外，更加明显的是在使用 `VAR` 的地方都添加了 `unsafe` 代码块。为什么？所有的线程都能访问 `VAR` ，且它是可以被修改的，自然就是不安全的。上面的代码比较简单，同一时间只会有一个线程读写 `VAR` ，不会有什么问题，所以用 `unsafe` 来标记就可以。如果是更多的线程，还是请使用接下来要介绍的同步机制来处理。

static如此，那const呢？ const会在编译时内联到代码中，所以不会存在某个固定的内存地址上，也不存在可以修改的情况，并不是内存共享的。

堆

由于现代操作系统的设计，线程寄生于进程，可以共享进程的资源，如果要在各个线程中共享一个变量，那么除了上面的static，还有就是把变量保存在堆上了。当然Rust也不例外，遵从这一设计。只是我们知道Rust在安全性上肯定又会做一些考量，从而在语言设计和使用上稍有不同。

为了在堆上分配空间，Rust提供了 `std::boxed::Box` ，由于堆的特点，存活时间比较长，所以除了我们这个地方介绍的线程间共享外，还有其他的用处，此处不详细说明，若不甚了解，请学习或回顾堆、栈与**Box**章节的介绍。下面我们来看一下如何在多个线程间访问 `Box` 创建的变量：

```
use std::thread;
use std::sync::Arc;

fn main() {
    let var : Arc<i32> = Arc::new(5);
    let share_var = var.clone();

    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        println!("share value in new thread: {}, address: {:p}",
 share_var, &*share_var);
    });

    // 等待新建线程先执行
    new_thread.join().unwrap();
    println!("share value in main thread: {}, address: {:p}", va
r, &*var);
}
```

运行结果：

```
share value in new thread: 5, address: 0x2825070
share value in main thread: 5, address: 0x2825070
```

你可能会觉得很奇怪，上面怎么没有看到Box创建的变量啊，这明明就是 `Arc` 的使用呀？ `Box` 创建的变量要想在多个线程中安全使用，我们还需要实现很多功能才行，需要是 `Sync` ，而 `Arc` 正是利用 `Box` 来实现的一个通过引用计数来共享状态的包裹类。下面引用一段 `Arc::new` 的源码即可看出它是通过 `Box` 来实现的：

```
pub fn new(data: T) -> Arc<T> {
    // Start the weak pointer count as 1 which is the weak point
er that's
    // held by all the strong pointers (kinda), see std/rc.rs fo
r more info
    let x: Box<_> = box ArcInner {
        strong: atomic::AtomicUsize::new(1),
        weak: atomic::AtomicUsize::new(1),
        data: data,
    };
    Arc { _ptr: unsafe { NonZero::new(Box::into_raw(x)) } }
}
```

通过上面的运行结果，我们也可以发现新建线程和主线程中打印的 `address` 是一样的，说明状态确实是在同一个内存地址处。

如果 `Box` 在堆上分配的资源仅在一个线程中使用，那么释放时，就非常简单，使用完，及时释放即可。如果是要在多个线程中使用，就需要面临两个关键问题：

1. 资源何时释放？
2. 线程如何安全的并发修改和读取？

由于上面两个问题的存在，这就是为什么我们不能直接用 `Box` 变量在线程中共享的原因，可以看出来，共享内存比消息传递机制似乎要复杂许多。Rust用了引用计数的方式来解决第一个问题，在标准库中提供了两个包裹类，除了上面一个用于多线程的 `std::sync::Arc` 之外，还有一个不能用于多线程的 `std::rc::Rc` 。在使用时，可以根据需要进行选择。如果你一不小心把 `std::rc::Rc` 用于多线程中，编译器会毫不客气地纠正你的。

关于上面的第二个问题，Rust语言及标准库提供了一系列的同步手段来解决。下面的章节我们将详细讲解这些方式和用法。

同步

同步指的是线程之间的协作配合，以共同完成某个任务。在整个过程中，需要注意两个关键点：一是共享资源的访问， 二是访问资源的顺序。通过前面的介绍，我们已经知道了如何让多个线程访问共享资源，但并没介绍如何控制访问顺序，才不会出现错误。如果两个线程同时访问同一内存地址的数据，一个写，一个读，如果不加控制，写线程只写了一半，读线程就开始读，必然读到的数据是错误的，不可用的，从而造成程序错误，这就造成了并发安全问题，为此我们必须要有一套控制机制来避免这样的事情发生。就好比两个人喝一瓶可乐，只有一根吸管，那肯定也得商量出一个规则，才能相安无事地都喝到可乐。本节就将具体介绍在Rust中，我们要怎么做，才能解决这个问题。

继续上面喝可乐的例子，一人一口的方式，就是一种解决方案，只要不是太笨，几乎都能想到这个方案。具体实施时，A在喝的时候，B一直在旁边盯着，要是A喝完一口，B马上拿过来喝，此时A肯定也是在旁边盯着。在现实生活中，这样的示例比比皆是。细想一下，貌似同步中都可能涉及到等待。诸葛先生在万事具备，只欠东风时，也只能等，因为条件不成熟啊。依照这个逻辑，在操作系统和各大编程语言中，几乎都支持当前线程等待，当然Rust也不例外。

等待

Rust中线程等待和其他语言在机制上并无差异，大致有下面几种：

- 等待一段时间后，再接着继续执行。看起来就像一个人工作累了，休息一会再工作。通过调用相关的API可以让当前线程暂停执行进入睡眠状态，此时调度器不会调度它执行，等过一段时间后，线程自动进入就绪状态，可以被调度执行，继续从之前睡眠时的地方执行。对应的API有 `std::thread::sleep` ， `std::thread::sleep_ms` ， `std::thread::park_timeout` ， `std::thread::park_timeout_ms` ，还有一些类似的其他API，由于太多，详细信息就请参见官网 `std::thread` 。
- 这一种方式有点特殊，时间非常短，就一个时间片，当前线程自己主动放弃当前时间片的调度，让调度器重新选择线程来执行，这样就把运行机会给了别的线程，但是要注意的是，如果别的线程没有更好的理由执行，当然最后执行机会还是它的。在实际的应用业务中，比如生产者制造出一个产品后，可以放弃一个时间片，让消费者获得执行机会，从而快速地消费才生产的产品。这样的

控制粒度非常小，需要合理使用，如果需要连续放弃多个时间片，可以借用循环实现。对应的API是 `std::thread::yield_now` ，详细信息参见官网 `std::thread` 。
- 1和2的等待都无须其他线程的协助，即可在一段时间后继续执行。最后我们还遇到一种等待，是需要其他线程参与，才能把等待的线程叫醒，否则，线程会一直等待下去。好比一个女人，要是没有遇到一个男人，就永远不可能摆脱单身的状态。相关的API包括 `std::thread::JoinHandle::join` ， `std::thread::park` ， `std::sync::Mutex::lock` 等，还有一些同步相关的类的API也会导致线程等待。详细信息参见官网 `std::thread` 和 `std::sync` 。

第一种和第三种等待方式，其实我们在上面的介绍中，都已经遇到过了，它们也是使用的最多的两种方式。在此，也可以回过头去看看前面的使用方式和使用效果，结合自己的理解，做一些简单的练习。

毫无疑问，第三种方式稍显复杂，要将等待的线程叫醒，必然基于一定的规则，比如早上7点必须起床，那么就定一个早上7点的闹钟，到时间了就响，没到时间别响。不管基于什么规则，要触发叫醒这个事件，就肯定是某个条件已经达成了。基于这样的逻辑，在操作系统和编程语言中，引入了一种叫着条件变量的东西。可以模拟现实生活中的闹钟的行为，条件达成就通知等待条件的线程。Rust的条件变量就是 `std::sync::Condvar` ，详情参见官网条件变量。但是通知也并不只是条件变量的专利，还有其他的方式也可以触发通知，下面我们就来瞧一瞧。

通知

看是简单的通知，在编程时也需要注意以下几点：

- 通知必然是因为有等待，所以通知和等待几乎都是成对出现的，比如 `std::sync::Condvar::wait` 和 `std::sync::Condvar::notify_one` ， `std::sync::Condvar::notify_all` 。
- 等待所使用的对象，与通知使用的对象是同一个对象，从而该对象需要在多个线程之间共享，参见下面的例子。
- 除了 `Condvar` 之外，其实锁也是具有自动通知功能的，当持有锁的线程释放锁的时候，等待锁的线程就会自动被唤醒，以抢占锁。关于锁的介绍，在下面有详解。
- 通过条件变量和锁，还可以构建更加复杂的自动通知方式，比如 `std::sync::Barrier` 。

- 通知也可以是1:1的，也可以是1:N的， `Condvar` 可以控制通知一个还是N个，而锁则不能控制，只要释放锁，所有等待锁的其他线程都会同时醒来，而不是只有最先等待的线程。

下面我们分析一个简单的例子：

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {

    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    // 创建一个新线程
    thread::spawn(move|| {
        let &(ref lock, ref cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();
        println!("notify main thread");
    });

    // 等待新线程先运行
    let &(ref lock, ref cvar) = &*pair;
    let mut started = lock.lock().unwrap();
    while !*started {
        println!("before wait");
        started = cvar.wait(started).unwrap();
        println!("after wait");
    }
}
```

运行结果：

```
before wait
notify main thread
after wait
```

这个例子展示了如何通过条件变量和锁来控制新建线程和主线程的同步，让主线程等待新建线程执行后，才能继续执行。从结果来看，功能上是实现了。对于上面这个例子，还有下面几点需要说明：

- `Mutex` 是Rust中的一种锁。
- `Condvar` 需要和 `Mutex` 一同使用，因为有 `Mutex` 保护， `Condvar` 并发才是安全的。
- `Mutex::lock` 方法返回的是一个 `MutexGuard` ，在离开作用域的时候，自动销毁，从而自动释放锁，从而避免锁没有释放的问题。
- `Condvar` 在等待时，时会释放锁的，被通知唤醒时，会重新获得锁，从而保证并发安全。

到此，你应该对锁比较感兴趣了，为什么需要锁？锁存在的目的就是为了保证资源在同一个时间，能有序地被访问，而不会出现异常数据。但其实要做到这一点，也并不是只有锁，包括锁在内，主要涉及两种基本方式：

原子类型

原子类型是最简单的控制共享资源访问的一种机制，相比较于后面将介绍的锁而言，原子类型不需要开发者处理加锁和释放锁的问题，同时支持修改，读取等操作，还具备较高的并发性能，从硬件到操作系统，到各个语言，基本都支持。在标准库 `std::sync::atomic` 中，你将在里面看到Rust现有的原子类型，包括 `AtomicBool` ， `AtomicIsize` ， `AtomicPtr` ， `AtomicUsize` 。这4个原子类型基本能满足百分之九十的共享资源安全访问的需要。下面我们就用原子类型，结合共享内存的知识，来展示一下一个线程修改，一个线程读取的情况：

```
use std::thread;
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};

fn main() {
    let var : Arc<AtomicUsize> = Arc::new(AtomicUsize::new(5));
    let share_var = var.clone();

    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        println!("share value in new thread: {}", share_var.load
(Ordering::SeqCst));
        // 修改值
        share_var.store(9, Ordering::SeqCst);
    });

    // 等待新建线程先执行
    new_thread.join().unwrap();
    println!("share value in main thread: {}", var.load(Ordering
::SeqCst));
}
```

运行结果：

```
share value in new thread: 5
share value in main thread: 9
```

结果表明新建线程成功的修改了值，并在主线程中获取到了最新值，你也可以尝试使用其他的原子类型。此处我们可以思考一下，如果我们用 `Arc::new(*mut Box<u32>)` 是否也可以做到？ 为什么？ 思考后，大家将体会到Rust在多线程安全方面做的有多么的好。除了原子类型，我们还可以使用锁来实现同样的功能。

锁

在多线程中共享资源，除了原子类型之外，还可以考虑用锁来实现。在操作之前必须先获得锁，一把锁同时只能给一个线程，这样能保证同一时间只有一个线程能操作共享资源，操作完成后，再释放锁给等待的其他线程。在Rust

中 `std::sync::Mutex` 就是一种锁。下面我们用 `Mutex` 来实现一下上面的原子类型的例子：

```
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    let var : Arc<Mutex<u32>> = Arc::new(Mutex::new(5));
    let share_var = var.clone();

    // 创建一个新线程
    let new_thread = thread::spawn(move|| {
        let mut val = share_var.lock().unwrap();
        println!("share value in new thread: {}", *val);
        // 修改值
        *val = 9;
    });

    // 等待新建线程先执行
    new_thread.join().unwrap();
    println!("share value in main thread: {}", *(var.lock().unwr
ap()));
}
```

运行结果：

```
share value in new thread: 5
share value in main thread: 9
```

结果都一样，看来用 `Mutex` 也能实现，但如果从效率上比较，原子类型会更胜一筹。暂且不论这点，我们从代码里面看到，虽然有 `lock` ，但是并么有看到有类似于 `unlock` 的代码出现，并不是不需要释放锁，而是Rust为了提高安全性，已然在 `val` 销毁的时候，自动释放锁了。同时我们发现，为了修改共享的值，开发者必须要调用 `lock` 才行，这样就又解决了一个安全问题。不得不再次赞叹一下Rust在多线程方面的安全性做得真是太好了。如果是其他语言，我们要做到安全，必然得自己来实现这些。

为了保障锁使用的安全性问题，Rust做了很多工作，但从效率来看还不如原子类型，那么锁是否就没有存在的价值了？显然事实不可能是这样的，既然存在，那必然有其价值。它能解决原子类型锁不能解决的那百分之十的问题。我们再来看一下之前的一个例子：

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {

    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    // 创建一个新线程
    thread::spawn(move|| {
        let &(ref lock, ref cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();
        println!("notify main thread");
    });

    // 等待新线程先运行
    let &(ref lock, ref cvar) = &*pair;
    let mut started = lock.lock().unwrap();
    while !*started {
        println!("before wait");
        started = cvar.wait(started).unwrap();
        println!("after wait");
    }
}
```

代码中的 `Condvar` 就是条件变量，它提供了 `wait` 方法可以主动让当前线程等待，同时提供了 `notify_one` 方法，让其他线程唤醒正在等待的线程。这样就能完美实现顺序控制了。看起来好像条件变量把事都做完了，要 `Mutex` 干嘛呢？为了防止多个线程同时执行条件变量的 `wait` 操作，因为条件变量本身也是需要被保护的，这就是锁能做，而原子类型做不到的地方。

在Rust中， `Mutex` 是一种独占锁，同一时间只有一个线程能持有这个锁。这种锁会导致所有线程串行起来，这样虽然保证了安全，但效率并不高。对于写少读多的情况来说，如果在没有写的情况下，都是读取，那么应该是可以并发执行的，为了达到这个目的，几乎所有的编程语言都提供了一种叫读写锁的机制，Rust中也存在，叫 `std::sync::RwLock` ，在使用上同 `Mutex` 差不多，在此就留给大家自行练习了。

同步是多线程编程的永恒主题，Rust已经为我们提供了良好的编程范式，并强加检查，即使你之前没有怎么接触过，用Rust也能编写出非常安全的多线程程序。

并行

理论上并行和语言并没有什么关系，所以在理论上的并行方式，都可以尝试用Rust来实现。本小节不会详细全面地介绍具体的并行理论知识，只介绍用Rust如何来实现相关的并行模式。

Rust的一大特点是，可以保证“线程安全”。而且，没有性能损失。更有意思的是，Rust编译器实际上只有 `Send` `Sync` 等基本抽象，而对“线程” “锁” “同步” 等基本的并行相关的概念一无所知，这些概念都是由库实现的。这意味着Rust实现并行编程可以有比较好的扩展性，可以很轻松地用库来支持那些常见的并行编程模式。 下面，我们以一个例子来演示一下，Rust如何将线程安全/执行高效/使用简单结合起来的。

在图形编程中，我们经常要处理归一化的问题： 即把一个范围内的值，转换到范围1内的值。比如把一个颜色值255归一后就是1。假设我们有一个表示颜色值的数组要进行归一，用非并行化的方式来处理非常简单，可以自行尝试。下面我们将采用并行化的方式来处理，把数组中的值同时分开给多个线程一起并行归一化处理。

```
extern crate rayon;

use rayon::prelude::*;

fn main() {
    let mut colors = [-20.0f32, 0.0, 20.0, 40.0,
        80.0, 100.0, 150.0, 180.0, 200.0, 250.0, 300.0];
    println!("original:    {:?}", &colors);

    colors.par_iter_mut().for_each(|color| {
        let c : f32 = if *color < 0.0 {
                0.0
            } else if *color > 255.0 {
                255.0
            } else {
                *color
            };
        *color = c / 255.0;
    });
    println!("transformed: {:?}", &colors);
}
```

运行结果：

```
original:    [-20, 0, 20, 40, 80, 100, 150, 180, 200, 250, 300]
transformed: [0, 0, 0.078431375, 0.15686275, 0.3137255, 0.392156
87, 0.5882353, 0.7058824, 0.78431374, 0.98039216, 1]
```

以上代码是不是很简单。调用 `par_iter_mut` 获得一个并行执行的具有写权限的迭代器， `for_each` 对每个元素执行一个操作。仅此而已。 我们能这么轻松地完成这个任务，原因是我们引入了 rayon 这个库。它把所有的脏活累活都干完了，把清晰安全易用的接口暴露出来给了我们。Rust还可以完全以库的形式，实现异步IO、协程等更加高阶的并行程序开发模式。

为了更深入的加深对Rust并发编程的理解和实践，还安排了一个挑战任务：实现一个Rust版本的MapReduce模式。值得你挑战。

Unsafe、原始指针

本章开始讲解 Rust 中的 `Unsafe` 部分。

unsafe

Rust的内存安全依赖于强大的类型系统和编译时检测，不过它并不能适应所有的场景。 首先，所有的编程语言都需要跟外部的“不安全”接口打交道，调用外部库等，在“安全”的Rust下是无法实现的; 其次，“安全”的Rust无法高效表示复杂的数据结构，特别是数据结构内部有各种指针互相引用的时候；再次， 事实上还存在着一些操作，这些操作是安全的，但不能通过编译器的验证。

因此在安全的Rust背后，还需要 `unsafe` 的支持。

`unsafe` 块能允许程序员做的额外事情有：

- 解引用一个裸指针 `*const T` 和 `*mut T`

```
let x = 5;
let raw = &x as *const i32;
let points_at = unsafe { *raw };
println!("raw points at {}", points_at);
```

- 读写一个可变的静态变量 `static mut`

```
static mut N: i32 = 5;
unsafe {
    N += 1;
    println!("N: {}", N);
}
```

- 调用一个不安全函数

```
unsafe fn foo() {
    //实现
}
fn main() {
    unsafe {
        foo();
    }
}
```

使用 `unsafe`

`unsafe fn` 不安全函数标示如果调用它可能会违反**Rust**的内存安全语意：

```
unsafe fn danger_will_robinson() {
    // 实现
}
```

`unsafe block` 不安全块可以在其中调用不安全的代码：

```
unsafe {
    // 实现
}
```

`unsafe trait` 不安全trait及它们的实现，所有实现它们的具体类型有可能是不安全的:

```
unsafe trait Scary { }
unsafe impl Scary for i32 {}
```

safe != no bug

对于**Rust**来说禁止你做任何不安全的事是它的本职，不过有些是编写代码时的 `bug` ，它们并不属于“内存安全”的范畴：

- 死锁

- 内存或其他资源溢出
- 退出未调用析构函数
- 整型溢出

使用 `unsafe` 时需要注意一些特殊情形：

- 数据竞争
- 解引用空裸指针和悬垂裸指针
- 读取未初始化的内存
- 使用裸指针打破指针重叠规则
- `&mut T` 和 `&T` 遵循LLVM范围的 `noalias` 模型，除了如果 `&T` 包含一个 `UnsafeCell<U>` 的话。不安全代码必须不能违反这些重叠（aliasing）保证
- 不使用 `UnsafeCell<U>` 改变一个不可变值/引用
- 通过编译器固有功能调用未定义行为：
  - 使用 `std::ptr::offset` （offset功能）来索引超过对象边界的值，除了允许的末位超出一个字节
  - 在重叠（overlapping）缓冲区上使用 `std::ptr::copy_nonoverlapping_memory` （memcpy32/memcpy64功能）
- 原生类型的无效值，即使是在私有字段/本地变量中：
  - 空/悬垂引用或装箱
  - `bool` 中一个不是 `false` （0）或 `true` （1）的值
  - `enum` 中一个并不包含在类型定义中判别式
  - `char` 中一个代理字（surrogate）或超过char::MAX的值
  - `str` 中非UTF-8字节序列
- 在外部代码中使用Rust或在Rust中使用外部语言

裸指针

Rust通过限制智能指针的行为保障了编译时安全，不过仍需要对指针做一些额外的操作。

`*const T` 和 `*mut T` 在**Rust**中被称为“裸指针”。它允许别名，允许用来写共享所有权的类型，甚至是内存安全的共享内存类型如：`Rc<T>` 和 `Arc<T>` ，但是赋予你更多权利的同时意味着你需要担当更多的责任：

- 不能保证指向有效的内存，甚至不能保证是非空的
- 没有任何自动清除，所以需要手动管理资源
- 是普通旧式类型，也就是说，它不移动所有权，因此**Rust**编译器不能保证不出像释放后使用这种bug
- 缺少任何形式的生命周期，不像 `&` ，因此编译器不能判断出悬垂指针
- 除了不允许直接通过 `*const T` 改变外，没有别名或可变性的保障

使用

创建一个裸指针：

```
let a = 1;
let b = &a as *const i32;

let mut x = 2;
let y = &mut x as *mut i32;
```

解引用需要在 `unsafe` 中进行：

```
let a = 1;
let b = &a as *const i32;
let c = unsafe { *b };
println!("{}", c);
```

`Box<T>` 的 `into_raw` ：

```
let a: Box<i32> = Box::new(10);
// 我们需要先解引用a，再隐式把 & 转换成 *
let b: *const i32 = &*a;
// 使用 into_raw 方法
let c: *const i32 = Box::into_raw(a);
```

如上说所，引用和裸指针之间可以隐式转换，但隐式转换后再解引用需要使用 `unsafe`：

```
// 显式
let a = 1;
let b: *const i32 = &a as *const i32; //或者let b = &a as *const i32；
// 隐式
let c: *const i32 = &a;
unsafe {
    println!("{}", *c);
}
```

FFI

FFI(Foreign Function Interface)是用来与其它语言交互的接口，在有些语言里面称为语言绑定(language bindings)，Java 里面一般称为 JNI(Java Native Interface) 或 JNA(Java Native Access)。由于现实中很多程序是由不同编程语言写的，必然会涉及到跨语言调用，比如 A 语言写的函数如果想在 B 语言里面调用，这时一般有两种解决方案：一种是将函数做成一个服务，通过进程间通信(IPC)或网络协议通信(RPC, RESTful等)；另一种就是直接通过 FFI 调用。前者需要至少两个独立的进程才能实现，而后者直接将其它语言的接口内嵌到本语言中，所以调用效率比前者高。

当前的系统编程领域大部分被 C/C++ 占领，而 Rust 定位为系统编程语言，少不了与现有的 C/C++ 代码交互，另外为了给那些"慢"脚本语言调用，Rust 必然得对 FFI 有完善的支持，本章我们就来谈谈 Rust 的 FFI 系统。

调用ffi函数

> 下文提到的ffi皆指cffi。

Rust作为一门系统级语言，自带对ffi调用的支持。

Getting Start

引入libc库

由于 `cffi` 的数据类型与 `rust` 不完全相同，我们需要引入 `libc` 库来表达对应 `ffi` 函数中的类型。

在 `Cargo.toml` 中添加以下行:

```
[dependencies]
libc = "0.2.9"
```

在你的rs文件中引入库:

```
extern crate libc
```

在以前 `libc` 库是和 `rust` 一起发布的，后来libc被移入了 `crates.io` 通过cargo安装。

声明你的 `ffi` 函数

就像 `c语言` 需要 `#include` 声明了对应函数的头文件一样， `rust` 中调用 `ffi` 也需要对对应函数进行声明。

```
use libc::c_int;
use libc::c_void;
use libc::size_t;

#[link(name = "yourlib")]
extern {
    fn your_func(arg1: c_int, arg2: *mut c_void) -> size_t; //
声明ffi函数
    fn your_func2(arg1: c_int, arg2: *mut c_void) -> size_t;
    static ffi_global: c_int; // 声明ffi全局变量
}
```

声明一个 `ffi` 库需要一个标记有 `#[link(name = "yourlib")]` 的 `extern` 块。 `name` 为对应的库(`so` / `dll` / `dylib` / `a`)的名字。 如：如果你需要 `snappy` 库(`libsnappy.so` / `libsnappy.dll` / `libsnappy.dylib` / `libsnappy.a`), 则对应的 `name` 为 `snappy` 。 在一个 `extern块` 中你可以声明任意多的函数和变量。

调用ffi函数

声明完成后就可以进行调用了。 由于此函数来自外部的c库，所以rust并不能保证该函数的安全性。因此，调用任何一个 `ffi` 函数需要一个 `unsafe` 块。

```
let result: size_t = unsafe {
    your_func(1 as c_int, Box::into_raw(Box::new(3)) as *mut c_v
oid)
};
```

封装 `unsafe` ，暴露安全接口

作为一个库作者，对外暴露不安全接口是一种非常不合格的做法。在做c库的 `rust binding` 时，我们做的最多的将是将不安全的c接口封装成一个安全接口。 通常做法是：在一个叫 `ffi.rs` 之类的文件中写上所有的 `extern块` 用以声明ffi函数。在一个叫 `wrapper.rs` 之类的文件中进行包装：

```
// ffi.rs
#[link(name = "yourlib")]
extern {
    fn your_func(arg1: c_int, arg2: *mut c_void) -> size_t;
}
```

```
// wrapper.rs
fn your_func_wrapper(arg1: i32, arg2: &mut i32) -> isize {
    unsafe { your_func(1 as c_int, Box::into_raw(Box::new(3)) as
 *mut c_void) } as isize
}
```

对外暴露(pub use) `your_func_wrapper` 函数即可。

数据结构对应

`libc` 为我们提供了很多原始数据类型，比如 `c_int` , `c_float` 等，但是对于自定义类型，如结构体，则需要我们自行定义。

结构体

`rust` 中结构体默认的内存表示和C并不兼容。如果要将结构体传给ffi函数，请为 `rust` 的结构体打上标记：

```
#[repr(C)]
struct RustObject {
    a: c_int,
    // other members
}
```

此外，如果使用 `#[repr(C, packed)]` 将不为此结构体填充空位用以对齐。

Union

比较遗憾的是，rust到目前为止(2016-03-31)还没有一个很好的应对c的union的方法。只能通过一些hack来实现。(对应rfc)

Enum

和 `struct` 一样，添加 `#[repr(C)]` 标记即可。

回调函数

和c库打交道时，我们经常会遇到一个函数接受另一个回调函数的情况。将一个 `rust` 函数转变成c可执行的回调函数非常简单：在函数前面加上 `extern "C"` :

```
extern "C" fn callback(a: c_int) { // 这个函数是传给c调用的
    println!("hello {}!", a);
}

#[link(name = "yourlib")]
extern {
   fn run_callback(data: i32, cb: extern fn(i32));
}

fn main() {
    unsafe {
        run_callback(1 as i32, callback); // 打印 1
    }
}
```

对应c库代码:

```
typedef void (*rust_callback)(int32_t);

void run_callback(int32_t data, rust_callback callback) {
    callback(data); // 调用传过来的回调函数
}
```

字符串

rust为了应对不同的情况，有很多种字符串类型。其中 `CStr` 和 `CString` 是专用于 `ffi` 交互的。

CStr

对于产生于c的字符串(如在c程序中使用 `malloc` 产生)，rust使用 `CStr` 来表示，和 `str` 类型对应，表明我们并不拥有这个字符串。

```
use std::ffi::CStr;
use libc::c_char;
#[link(name = "yourlib")]
extern {
    fn char_func() -> *mut c_char;
}

fn get_string() -> String {
    unsafe {
        let raw_string: *mut c_char = char_func();
        let cstr = CStr::from_ptr(my_string());
        cstr.to_string_lossy().into_owned()
    }
}
```

在这里 `get_string` 使用 `CStr::from_ptr` 从c的 `char*` 获取一个字符串，并且转化成了一个String.

- 注意to_string_lossy()的使用：因为在rust中一切字符都是采用utf8表示的而c不是，因此如果要将c的字符串转换到rust字符串的话，需要检查是否都为有效 `utf-8` 字节。 `to_string_lossy` 将返回一个 `Cow<str>` 类型，即如果c字符串都为有效 `utf-8` 字节，则将其0开销地转换成一个 `&str` 类型，若不是，rust会将其拷贝一份并且将非法字节用 `U+FFFD` 填充。

CString

和 `CStr` 表示从c中来，rust不拥有归属权的字符串相反， `CString` 表示由rust分配，用以传给c程序的字符串。

```
use std::ffi::CString;
use std::os::raw::c_char;

extern {
    fn my_printer(s: *const c_char);
}

let c_to_print = CString::new("Hello, world!").unwrap();
unsafe {
    my_printer(c_to_print.as_ptr()); // 使用 as_ptr 将CString转化成char指针传给c函数
}
```

注意c字符串中并不能包含 `\0` 字节(因为 `\0` 用来表示c字符串的结束符),因此 `CString::new` 将返回一个 `Result` ， 如果输入有 `\0` 的话则为 `Error(NulError)` 。

不透明结构体

C库存在一种常见的情况：库作者并不想让使用者知道一个数据类型的具体内容，因此常常提供了一套工具函数，并使用 `void*` 或不透明结构体传入传出进行操作。 比较典型的是 `ncurse` 库中的 `WINDOW` 类型。

当参数是 `void*` 时，在rust中可以和c一样，使用对应类型 `*mut libc::c_void` 进行操作。如果参数为不透明结构体，rust中可以使用空白 `enum` 进行代替:

```
enum OpaqueStruct {}

extern "C" {
    pub fn foo(arg: *mut OpaqueStruct);
}
```

C代码：

```
struct OpaqueStruct;
void foo(struct OpaqueStruct *arg);
```

空指针

另一种很常见的情况是需要一个空指针。请使用 `0 as *const _` 或者 `std::ptr::null()` 来生产一个空指针。

内存安全

由于 `ffi` 跨越了rust边界，rust编译器此时无法保障代码的安全性，所以在涉及ffi操作时要格外注意。

析构问题

在涉及ffi调用时最常见的就是析构问题：这个对象由谁来析构？是否会泄露或use after free？ 有些情况下c库会把一类类型 `malloc` 了以后传出来，然后不再关系它的析构。因此在做ffi操作时请为这些类型实现析构(`Drop Trait`).

可空指针优化

当 `rust` 的一个 `enum` 为一种特殊结构：它有两种实例，一种为空，另一种只有一个数据域的时候，rustc会开启空指针优化将其优化成一个指针。 比如 `Option<extern "C" fn(c_int) -> c_int>` 会被优化成一个可空的函数指针。

ownership处理

在rust中，由于编译器会自动插入析构代码到块的结束位置，在使用 `owned` 类型时要格外的注意。

```
extern {
    pub fn foo(arg: extern fn() -> *const c_char);
}

extern "C" fn danger() -> *const c_char {
    let cstring = CString::new("I'm a danger string").unwrap();
    cstring.as_ptr()
}  // 由于CString是owned类型，在这里cstring被rust free掉了。USE AFTE
R FREE! too young!

fn main() {
  unsafe {
        foo(danger); // boom !!
    }
}
```

由于 `as_ptr` 接受一个 `&self` 作为参数(`fn as_ptr(&self) -> *const c_char`)， `as_ptr` 以后 `ownership` 仍然归rust所有。因此rust会在函数退出时进行析构。 正确的做法是使用 `into_raw()` 来代替 `as_ptr()` 。由于 `into_raw` 的签名为 `fn into_raw(self) -> *mut c_char` ，接受的是 `self` ,产生了 `ownership` 转移， 因此 `danger` 函数就不会将 `cstring` 析构了。

panic

由于在 `ffi` 中 `panic` 是未定义行为，切忌在 `cffi` 时 `panic` 包括直接调用 `panic!` , `unimplemented!` ,以及强行 `unwrap` 等情况。 当你写 `cffi` 时，记住：你写下的每个单词都可能是发射核弹的密码！

静态库/动态库

前面提到了声明一个外部库的方式-- `#[link]` 标记，此标记默认为动态库。但如果是静态库，可以使用 `#[link(name = "foo", kind = "static")]` 来标记。此外，对于osx的一种特殊库-- `framework` , 还可以这样标记 `#[link(name = "CoreFoundation", kind = "framework")]` .

调用约定

前面看到，声明一个被c调用的函数时，采用 `extern "C" fn` 的语法。此处的 `"C"` 即为c调用约定的意思。此外，rust还支持：

- stdcall
- aapcs
- cdecl
- fastcall
- vectorcall //这种call约定暂时需要开启abi_vectorcall feature gate.
- Rust
- rust-intrinsic
- system
- C
- win64

bindgen

是不是觉得把一个个函数和全局变量在 `extern块` 中去声明，对应的数据结构去手动创建特别麻烦？没关系， `rust-bindgen` 来帮你搞定。 `rust-bindgen` 是一个能从对应c头文件自动生成函数声明和数据结构的工具。创建一个绑定只需要 `./bindgen [options] input.h` 即可。 项目地址

将Rust编译成库

上一章讲述了如何从rust中调用c库，这一章我们讲如何把rust编译成库让别的语言通过cffi调用。

调用约定和mangle

正如上一章讲述的，为了能让rust的函数通过ffi被调用，需要加上 `extern "C"` 对函数进行修饰。

但由于rust支持重载，所以函数名会被编译器进行混淆，就像c++一样。因此当你的函数被编译完毕后，函数名会带上一串表明函数签名的字符串。

比如： `fn test() {}` 会变成 `_ZN4test20hf06ae59e934e5641haaE` . 这样的函数名为ffi调用带来了困难，因此，rust提供了 `#[no_mangle]` 属性为函数修饰。 对于带有 `#[no_mangle]` 属性的函数，rust编译器不会为它进行函数名混淆。如：

```
#[no_mangle]
extern "C" fn test() {}
```

在nm中观察到为

```
...
00000000001a7820 T test
...
```

至此， `test` 函数将能够被正常的由 `cffi` 调用。

指定 `crate` 类型

`rustc` 默认编译产生 `rust` 自用的 `rlib` 格式库，要让 `rustc` 产生动态链接库或者静态链接库，需要显式指定。

1. 方法1: 在文件中指定。 在文件头加上 `#![crate_type = "foo"]` , 其中 `foo` 的可选类型有 `bin` , `lib` , `rlib` , `dylib` , `staticlib` .分别对应

可执行文件， 默认(将由 `rustc` 自己决定), `rlib` 格式，动态链接库，静态链接库。
2. 方法2: 编译时给rustc 传 `--crate-type` 参数。参数内容同上。
3. 方法3: 使用cargo，指定 `crate-type = ["foo"]` , `foo` 可选类型同1

小技巧: `Any`

由于在跨越 `ffi` 过程中， `rust` 类型信息会丢失，比如当用 `rust` 提供一个 `OpaqueStruct` 给别的语言时：

```
use std::mem::transmute;

#[derive(Debug)]
struct Foo<T> {
  t: T
}

#[no_mangle]
extern "C" fn new_foo_vec() -> *const c_void {
    Box::into_raw(Box::new(Foo {t: vec![1,2,3]})) as *const c_vo
id
}

#[no_mangle]
extern "C" fn new_foo_int() -> *const c_void {
    Box::into_raw(Box::new(Foo {t: 1})) as *const c_void
}

fn push_foo_element(t: &mut Foo<Vec<i32>>) {
    t.t.push(1);
}

#[no_mangle]
extern "C" fn push_foo_element_c(foo: *mut c_void){
    let foo2 = unsafe {
        &mut *(foo as *mut Foo<Vec<i32>>) // 这么确定是Foo<Vec<i32
>>? 万一foo是Foo<i32>怎么办？
    };
    push_foo_element(foo3);
}
```

以上代码中完全不知道 `foo` 是一个什么东西。安全也无从说起了，只能靠文档。因此在 `ffi` 调用时往往会丧失掉 `rust` 类型系统带来的方便和安全。在这里提供一个小技巧:使用 `Box<Box<Any>>` 来包装你的类型。

`rust` 的 `Any` 类型为 `rust` 带来了运行时反射的能力，使用 `Any` 跨越 `ffi` 边界将极大提高程序安全性。

```
use std::any::Any;

#[derive(Debug)]
struct Foo<T> {
  t: T
}

#[no_mangle]
extern "C" fn new_foo_vec() -> *const c_void {
    Box::into_raw(Box::new(Box::new(Foo {t: vec![1,2,3]}) as Box
<Any>)) as *const c_void
}

#[no_mangle]
extern "C" fn new_foo_int() -> *const c_void {
    Box::into_raw(Box::new(Box::new(Foo {t: 1}) as Box<Any>)) as
 *const c_void
}

fn push_foo_element(t: &mut Foo<Vec<i32>>) {
    t.t.push(1);
}

#[no_mangle]
extern "C" fn push_foo_element_c(foo: *mut c_void){
    let foo2 = unsafe {
        &mut *(foo as *mut Box<Any>)
    };
    let foo3: Option<&mut Foo<Vec<i32>>> = foo2.downcast_mut();
// 如果foo2不是*const Box<Foo<Vec<i32>>>, 则foo3将会是None
    if let Some(value) = foo3 {
      push_foo_element(value);
    }
}
```

这样一来，就非常不容易出错了。

运算符重载

Rust可以让我们对某些运算符进行重载，这其中大部分的重载都是对 `std::ops` 下的trait进行重载而实现的。

重载加法

我们现在来实现一个只支持加法的阉割版复数：

```
use std::ops::Add;

#[derive(Debug)]
struct Complex {
    a: f64,
    b: f64,
}

impl Add for Complex {
    type Output = Complex;
    fn add(self, other: Complex) -> Complex {
        Complex {a: self.a+other.a, b: self.b+other.b}
    }
}

fn main() {
    let cp1 = Complex{a: 1f64, b: 2.0};
    let cp2 = Complex{a: 5.0, b:8.1};
    let cp3 = cp1 + cp2;
    print!("{:?}", cp3);
}
```

输出:

```
Complex { a: 6, b: 10.1}
```

这里我们实现了 `std::ops::Add` 这个trait。这时候有同学一拍脑门，原来如此，没错……其实Rust的大部分运算符都是 `std::ops` 下的trait的语法糖！

我们来看看 `std::ops::Add` 的具体结构

```
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
    }
}
```

神奇的Output以及动态分发

有的同学会问了，这个 `Output` 是肿么回事？答，类型转换哟亲！ 举个不太恰当的栗子，我们在现实中会出现 `0.5+0.5=1` 这样的算式，用Rust的语言来描述就是： 两个 `f32` 相加得到了一个 `i8` 。显而易见，Output就是为这种情况设计的。

还是看代码：

```
use std::ops::Add;

#[derive(Debug)]
struct Complex {
    a: f64,
    b: f64,
}

impl Add for Complex {
    type Output = Complex;
    fn add(self, other: Complex) -> Complex {
        Complex {a: self.a+other.a, b: self.b+other.b}
    }
}

impl Add<i32> for Complex {
    type Output = f64;
    fn add(self, other: i32) -> f64 {
        self.a + self.b + (other as f64)
    }
}

fn main() {
    let cp1 = Complex{a: 1f64, b: 2.0};
    let cp2 = Complex{a: 5.0, b:8.1};
    let cp3 = Complex{a: 9.0, b:20.0};
    let complex_add_result = cp1 + cp2;
    print!("{:?}\n", complex_add_result);
    print!("{:?}", cp3 + 10i32);
}
```

输出结果：

```
Complex { a: 6, b: 10.1 }
39
```

对范型的限制

Rust的运算符是基于trait系统的，同样的，运算符可以被当成一种对范型的限制，我们可以这么要求 `范型T必须实现了trait Mul<Output=T>` 。于是，我们得到了如下的一份代码：

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Area of s: {}", s.area());
}
```

对于trait `HasArea<T>` 和 struct `Square<T>` ，我们通过 `where T: Mul<Output=T> + Copy` 限制了 `T` 必须实现乘法。同时Copy则限制了Rust不再将self.side给move到返回值里去。

写法简单，轻松愉快。

属性和编译器参数

本章将介绍Rust语言中的属性（Attribute）和编译器参数（Compiler Options）。

属性

属性（Attribute）是一种通用的用于表达元数据的特性，借鉴ECMA-334(C#)的语法来实现ECMA-335中描述的Attributes。属性只能应用于Item（元素、项），例如 `use` 声明、模块、函数等。

元素

在Rust中，Item是Crate（库）的一个组成部分。它包括

- `extern crate` 声明
- `use` 声明
- 模块（模块是一个Item的容器）
- 函数
- `type` 定义
- 结构体定义
- 枚举类型定义
- 常量定义
- 静态变量定义
- Trait定义
- 实现（Impl）

这些Item是可以互相嵌套的，比如在一个函数中定义一个静态变量、在一个模块中使用 `use` 声明或定义一个结构体。这些定义在某个作用域里面的Item跟你把 它写到最外层作用域所实现的功能是一样的，只不过你要访问这些嵌套的Item就必须使用路径（Path），如 `a::b::c` 。但一些外层的Item不允许你使用路径去 访问它的子Item，比如函数，在函数中定义的静态变量、结构体等，是不可以通过路径来访问的。

属性的语法

属性的语法借鉴于C#，看起来像是这样子的

```
#[name(arg1, arg2 = "param")]
```

它是由一个 `#` 开启，后面紧接着一个 `[]` ，里面便是属性的具体内容，它可以有如下几种写法：

- 单个标识符代表的属性名，如 `#[unix]`
- 单个标识符代表属性名，后面紧跟着一个 `=` ，然后再跟着一个字面量（Literal），组成一个键值对，如 `#[link(name = "openssl")]`
- 单个标识符代表属性名，后面跟着一个逗号隔开的子属性的列表，如 `#[cfg(and(unix, not(windows)))]`

在 `#` 后面还可以紧跟一个 `!` ，比如 `#![feature(box_syntax)]` ，这表示这个属性是应用于它所在的这个Item。而如果没有 `!` 则表示这个属性仅应用于紧接着的那个Item。

例如：

```
// 为这个crate开启box_syntax这个新特性
#![feature(box_syntax)]

// 这是一个单元测试函数
#[test]
fn test_foo() {
    /* ... */
}

// 条件编译，只会在编译目标为Linux时才会生效
#[cfg(target_os="linux")]
mod bar {
    /* ... */
}

// 为以下的这个type定义关掉non_camel_case_types的编译警告
#[allow(non_camel_case_types)]
type int8_t = i8;
```

应用于Crate的属性

- `crate_name` - 指定Crate的名字。如 `#[crate_name = "my_crate"]` 则可以让编译出的库名字为 `libmy_crate.rlib` 。

- `crate_type` - 指定Crate的类型，有以下几种选择
  - `"bin"` - 编译为可执行文件；
  - `"lib"` - 编译为库；
  - `"dylib"` - 编译为动态链接库；
  - `"staticlib"` - 编译为静态链接库；
  - `"rlib"` - 编译为Rust特有的库文件，它是一种特殊的静态链接库格式，它里面会含有一些元数据供编译器使用，最终会静态链接到目标文件之中。

  例 `#![crate_type = "dylib"]` 。
- `feature` - 可以开启一些不稳定特性，只可在nightly版的编译器中使用。
- `no_builtins` - 去掉内建函数。
- `no_main` - 不生成 `main` 这个符号，当你需要链接的库中已经定义了 `main` 函数时会用到。
- `no_start` - 不链接自带的 `native` 库。
- `no_std` - 不链接自带的 `std` 库。
- `plugin` - 加载编译器插件，一般用于加载自定义的编译器插件库。用法是

```
// 加载foo, bar两个插件
#![plugin(foo, bar)]
// 或者给插件传入必要的初始化参数
#![plugin(foo(arg1, arg2))]
```

- `recursive_limit` - 设置在编译期最大的递归层级。比如自动解引用、递归定义的宏等。默认设置是 `#![recursive_limit = "64"]`

应用于模块的属性

- `no_implicit_prelude` - 取消自动插入 `use std::prelude::*` 。
- `path` - 设置此 `mod` 的文件路径。

  如声明 `mod a;` ，则寻找
  - 本文件夹下的 `a.rs` 文件
  - 本文件夹下的 `a/mod.rs` 文件

```
#[cfg(unix)]
#[path = "sys/unix.rs"]
mod sys;

#[cfg(windows)]
#[path = "sys/windows.rs"]
mod sys;
```

应用于函数的属性

- `main` - 把这个函数作为入口函数，替代 `fn main` ，会被入口函数（Entry Point）调用。
- `plugin_registrar` - 编写编译器插件时用，用于定义编译器插件的入口函数。
- `start` - 把这个函数作为入口函数（Entry Point），改写 `start` language item。
- `test` - 指明这个函数为单元测试函数，在非测试环境下不会被编译。
- `should_panic` - 指明这个单元测试函数必然会panic。
- `cold` - 指明这个函数很可能是不会被执行的，因此优化的时候特别对待它。

```
// 把`my_main`作为主函数
#[main]
fn my_main() {

}

// 把`plugin_registrar`作为此编译器插件的入口函数
#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}

// 把`entry_point`作为入口函数，不再执行标准库中的初始化流程
#[start]
fn entry_point(argc: isize, argv: *const *const u8) -> isize {

}

// 定义一个单元测试
// 这个单元测试一定会panic
#[test]
#[should_panic]
fn my_test() {
    panic!("I expected to be panicked");
}

// 这个函数很可能是不会执行的，
// 所以优化的时候就换种方式
#[cold]
fn unlikely_to_be_executed() {

}
```

应用于全局静态变量的属性

- `thread_local` - 只可用于 `static mut`，表示这个变量是thread local的。

应用于FFI的属性

`extern` 块可以应用以下属性

- `link_args` - 指定链接时给链接器的参数，平台和实现相关。
- `link` - 说明这个块需要链接一个native库，它有以下参数：
    - `name` - 库的名字，比如 `libname.a` 的名字是 `name` ；
    - `kind` - 库的类型，它包括
        - `dylib` - 动态链接库
        - `static` - 静态库
        - `framework` - OS X里的Framework

    ```
    #[link(name = "readline")]
    extern {

    }

    #[link(name = "CoreFoundation", kind = "framework")]
    extern {

    }
    ```

在 `extern` 块里面，可以使用

- `link_name` - 指定这个链接的外部函数的名字或全局变量的名字；
- `linkage` - 对于全局变量，可以指定一些LLVM的链接类型（ http://llvm.org/docs/LangRef.html#linkage-types ）。

对于 `enum` 类型，可以使用

- `repr` - 目前接受 `C` ， `C` 表示兼容C ABI。

```
#[repr(C)]
enum eType {
    Operator,
    Indicator,
}
```

对于 `struct` 类型，可以使用

- `repr` - 目前只接受 `C` 和 `packed` ， `C` 表示结构体兼容C

ABI，`packed` 表示移除字段间的padding。

用于宏的属性

- `macro_use` - 把模块或库中定义的宏导出来
    - 应用于 `mod` 上，则把此模块内定义的宏导出到它的父模块中
    - 应用于 `extern crate` 上，则可以接受一个列表，如

    ```
    #[macro_use(debug, trace)]
    extern crate log;
    ```

    则可以只导入列表中指定的宏，若不指定则导入所有的宏。

- `macro_reexport` - 应用于 `extern crate` 上，可以再把这些导入的宏再输出出去给别的库使用。
- `macro_export` - 应于在宏上，可以使这个宏可以被导出给别的库使用。
- `no_link` - 应用于 `extern crate` 上，表示即使我们把它里面的库导入进来了，但是不要把这个库链接到目标文件中。

其它属性

- `export_function` - 用于静态变量或函数，指定它们在目标文件中的符号名。
- `link_section` - 用于静态变量或函数，表示应该把它们放到哪个段中去。
- `no_mangle` - 可以应用于任意的Item，表示取消对它们进行命名混淆，直接把它们的名字作为符号写到目标文件中。
- `simd` - 可以用于元组结构体上，并自动实现了数值运算符，这些操作会生成相应的SIMD指令。
- `doc` - 为这个Item绑定文档，跟 `///` 的功能一样，用法是

    ```
    #[doc = "This is a doc"]
    struct Foo {}
    ```

条件编译

有时候，我们想针对不同的编译目标来生成不同的代码，比如在编写跨平台模块时，针对Linux和Windows分别使用不同的代码逻辑。

条件编译基本上就是使用 `cfg` 这个属性，直接看例子

```
#[cfg(target_os = "macos")]
fn cross_platform() {
    // Will only be compiled on Mac OS, including Mac OS X
}

#[cfg(target_os = "windows")]
fn cross_platform() {
    // Will only be compiled on Windows
}

// 若条件`foo`或`bar`任意一个成立，则编译以下的Item
#[cfg(any(foo, bar))]
fn need_foo_or_bar() {

}

// 针对32位的Unix系统
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {

}

// 若`foo`不成立时编译
#[cfg(not(foo))]
fn needs_not_foo() {

}
```

其中，`cfg` 可接受的条件有

- `debug_assertions` - 若没有开启编译优化时就会成立。

- `target_arch = "..."` - 目标平台的CPU架构，包括但不限于 `x86` , `x86_64` , `mips` , `powerpc` , `arm` 或 `aarch64` 。
- `target_endian = "..."` - 目标平台的大小端，包括 `big` 和 `little` 。
- `target_env = "..."` - 表示使用的运行库，比如 `musl` 表示使用的是MUSL的libc实现, `msvc` 表示使用微软的MSVC， `gnu` 表示使用GNU的实现。 但在部分平台这个数据是空的。
- `target_family = "..."` - 表示目标操作系统的类别，比如 `windows` 和 `unix` 。这个属性可以直接作为条件使用，如 `#[unix]` ， `#[cfg(unix)]` 。
- `target_os = "..."` - 目标操作系统，包括但不限于 `windows` , `macos` , `ios` , `linux` , `android` , `freebsd` , `dragonfly` , `bitrig` , `openbsd` , `netbsd` 。
- `target_pointer_width = "..."` - 目标平台的指针宽度，一般就是 `32` 或 `64` 。
- `target_vendor = "..."` - 生产商，例如 `apple` , `pc` 或大多数Linux系统的 `unknown` 。
- `test` - 当启动了单元测试时（即编译时加了 `--test` 参数，或使用 `cargo test` ）。

还可以根据一个条件去设置另一个条件，使用 `cfg_attr` ，如

```
#[cfg_attr(a, b)]
```

这表示若 `a` 成立，则这个就相当于 `#[cfg(b)]` 。

条件编译属性只可以应用于Item，如果想应用在非Item中怎么办呢？可以使用 `cfg!` 宏，如

```
if cfg!(target_arch = "x86") {

} else if cfg!(target_arch = "x86_64") {

} else if cfg!(target_arch = "mips") {

} else {

}
```

这种方式不会产生任何运行时开销，因为不成立的条件相当于里面的代码根本不可能被执行，编译时会直接被优化掉。

Linter参数

目前的Rust编译器已自带的Linter，它可以在编译时静态帮你检测不用的代码、死循环、编码风格等等。Rust提供了一系列的属性用于控制Linter的行为

- `allow(C)` - 编译器将不会警告对于 `C` 条件的检查错误。
- `deny(C)` - 编译器遇到违反 `C` 条件的错误将直接当作编译错误。
- `forbit(C)` - 行为与 `deny(C)` 一样，但这个将不允许别人使用 `allow(C)` 去修改。
- `warn(C)` - 编译器将对于 `C` 条件的检查错误输出警告。

编译器支持的Lint检查可以通过执行 `rustc -W help` 来查看。

内联参数

内联函数即建议编译器可以考虑把整个函数拷贝到调用者的函数体中，而不是生成一个 `call` 指令调用过去。这种优化对于短函数非常有用，有利于提高性能。

编译器自己会根据一些默认的条件来判断一个函数是不是应该内联，若一个不应该被内联的函数被内联了，实际上会导致整个程序更慢。

可选的属性有：

- `#[inline]` - 建议编译器内联这个函数
- `#[inline(always)]` - 要求编译器必须内联这个函数

- `#[inline(never)]` - 要求编译器不要内联这个函数

内联会导致在一个库里面的代码被插入到另一个库中去。

自动实现Trait

编译器提供一个编译器插件叫作 `derive` ，它可以帮你去生成一些代码去实现（impl）一些特定的Trait，如

```
#[derive(PartialEq, Clone)]
struct Foo<T> {
    a: i32,
    b: T,
}
```

编译器会自动为你生成以下的代码

```
impl<T: PartialEq> PartialEq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }

    fn ne(&self, other: &Foo<T>) -> bool {
        self.a != other.a || self.b != other.b
    }
}

impl<T: Clone> Clone for Foo<T> {
    fn clone(&self) -> Foo<T> {
        Foo {
            a: self.a.clone(),
            b: self.b.clone(),
        }
    }
}
```

目前 `derive` 仅支持标准库中部分的Trait。

编译器特性

在非稳定版的Rust编译器中，可以使用一些不稳定的功能，比如一些还在讨论中的新功能、正在实现中的功能等。Rust编译器提供一个应用于Crate的属性 `feature` 来启用这些不稳定的功能，如

```
#![feature(advanced_slice_patterns, box_syntax, asm)]
```

具体可使用的编译器特性会因编译器版本的发布而不同，具体请阅读官方文档。

编译器参数

本章将介绍Rust编译器的参数。

Rust编译器程序的名字是 `rustc` ，使用它的方法很简单：

```
$ rustc [OPTIONS] INPUT
```

其中， `[OPTIONS]` 表示编译参数，而 `INPUT` 则表示输入文件。而编译参数有以下可选：

- `-h, --help` - 输出帮助信息到标准输出；
- `--cfg SPEC` - 传入自定义的条件编译参数，使用方法如

  ```
  fn main() {
      if cfg!(hello) {
          println!("world!");
      }
  }
  ```

  如上例所示，若 `cfg!(hello)` 成立，则运行程序就会输出 `"world"` 到标准输出。我们把这个文件保存为 `hello.rs` 然后编译它

  ```
  $ rustc --cfg hello hello.rs
  ```

  运行它就会看到屏幕中输出了 `world!` 。

- `-L [KIND=]PATH` - 往链接路径中加入一个文件夹，并且可以指定这个路径的类型（Kind），这些类型包括
  - `dependency` - 在这个路径下找依赖的文件，比如说 `mod` ；
  - `crate` - 只在这个路径下找 `extern crate` 中定义的库；
  - `native` - 只在这个路径下找Native库；
  - `framework` - 只在OS X下有用，只在这个路径下找Framework；
  - `all` - 默认选项。

- `-l [KIND=]NAME` - 链接一个库，这个库可以指定类型（Kind）
    - `static` - 静态库；
    - `dylib` - 动态库；
    - `framework` - OS X的Framework。

  如果不传，默认为 `dylib` 。

  此处举一个例子如何手动链接一个库，我们先创建一个文件叫 `myhello.rs` ，在里面写一个函数

```
// myhello.rs

/// 这个函数仅仅向标签输出打印 Hello World!
/// 不要忘记要把它标记为 pub 哦。
pub fn print_hello() {
    println!("Hello World!");
}
```

然后把这个文件编译成一个静态库， `libmyhello.a`

```
$ rustc --crate-type staticlib myhello.rs
```

然后再创建一个 `main.rs` ，链接这个库并打印出"Hello World!"

```
// main.rs

// 指定链接库 myhello
extern crate myhello;

fn main() {
    // 调用库函数
    myhello::print_hello();
}
```

编译 `main.rs`

```
$ rustc -L. -lmyhello main.rs
```

运行 `main` ，就会看到屏幕输出"Hello World!"啦。

- `--crate-type` - 指定编译输出类型，它的参数包括
    - `bin` - 二进行可执行文件
    - `lib` - 编译为库
    - `rlib` - Rust库
    - `dylib` - 动态链接库
    - `staticlib` - 静态链接库
- `--crate-name` - 指定这个Crate的名字，默认是文件名，如 `main.rs` 编译成可执行文件时默认是 `main` ，但你可以指定它为 `foo`

```
$ rustc --crate-name foo main.rs
```

则会输出 `foo` 可执行文件。

- `--emit` - 指定编译器的输出。编译器默认是输出一个可执行文件或库文件，但你可以选择输出一些其它的东西用于Debug
    - `asm` - 输出汇编
    - `llvm-bc` - LLVM Bitcode ；
    - `llvm-ir` - LLVM IR，即LLVM中间码（LLVM Intermediate Representation）；
    - `obj` - Object File（就是 `*.o` 文件）；
    - `link` - 这个是要结合其它 `--emit` 参数使用，会执行Linker再输出结果；
    - `dep-info` - 文件依赖关系（Debug用，类似于Makefile一样的依赖）。

以上参数可以同时使用，使用逗号分割，如

```
$ rustc --emit asm,llvm-ir,obj main.rs
```

同时，在最后可以加一个 `=PATH` 来指定输出到一个特定文件，如

```
$ rustc --emit asm=output.S,llvm-ir=output.ir main.rs
```

这样会把汇编生成到 `output.S` 文件中，把LLVM中间码输出到 `output.ir` 中。

- `--print` - 打印一些信息，参数有
  - `crate-name` - 编译目标名；
  - `file-names` - 编译的文件名；
  - `sysroot` - 打印Rust工具链的根目录地址。
- `-g` - 在目标文件中保存符号，这个参数等同于 `-C debuginfo=2` 。
- `-O` - 开启优化，这个参数等同于 `-C opt-level=2` 。
- `-o FILENAME` - 指定输出文件名，同样适用于 `--emit` 的输出。
- `--out-dir DIR` - 指定输出的文件夹，默认是当前文件夹，且会忽略 `-o` 配置。
- `--explain OPT` - 解释某一个编译错误，比如

  若你写了一个 `main.rs` ，使用了一个未定义变量 `f`

  ```
  fn main() {
      f
  }
  ```

  编译它时编译器会报错：

  ```
  main.rs:2:5: 2:6 error: unresolved name `f` [E0425]
  main.rs:2     f
                ^
  main.rs:2:5: 2:6 help: run `rustc --explain E0425` to see a
  detailed explanation
  error: aborting due to previous error
  ```

  虽然错误已经很明显，但是你也可以让编译器解释一下，什么是 `E0425` 错误：

  ```
  $ rustc --explain E0425
  // 编译器打印的说明
  ```

- `--test` - 编译成一个单元测试可执行文件

- `--target TRIPLE` - 指定目标平台，基本格式是 `cpu-manufacturer-kernel[-os]` ，例如

    ```
    ## 64位OS X
    $ rustc --target x86_64-apple-darwin
    ```

- `-W help` - 打印Linter的所有可配置选项和默认值。

- `-W OPT, --warn OPT` - 设置某一个Linter选项为Warning。

- `-A OPT, --allow OPT` - 设置某一个Linter选项为Allow。
- `-D OPT, --deny OPT` - 设置某一个Linter选项为Deny。
- `-F OPT, --forbit OPT` - 设置某一个Linter选项为Forbit。

- `-C FLAG[=VAL], --codegen FLAG[=VAL]` - 目标代码生成的的相关参数，可以用 `-C help` 来查看配置，值得关注的几个是

    - `linker=val` - 指定链接器；
    - `linker-args=val` - 指定链接器的参数；
    - `prefer-dynamic` - 默认Rust编译是静态链接，选择这个配置将改为动态链接；
    - `debug-info=level` - Debug信息级数， `0` = 不生成， `1` = 只生成文件行号表， `2` = 全部生成；
    - `opt-level=val` - 优化级数，可选 `0-3` ；
    - `debug_assertion` - 显式开启 `cfg(debug_assertion)` 条件。
- `-V, --version` - 打印编译器版本号。

- `-v, --verbose` - 开启啰嗦模式（打印编译器执行的日志）。

- `--extern NAME=PATH` - 用来指定外部的Rust库（ `*.rlib` ）名字和路径，名字应该与 `extern crate` 中指定的一样。

- `--sysroot PATH` - 指定工具链根目录。

- `-Z flag` - 编译器Debug用的参数，可以用 `-Z help` 来查看可用参数。

- `--color auto|always|never` - 输出时对日志加颜色

    - `auto` - 自动选择加还是不加，如果输出目标是虚拟终端（TTY）的话就加，否则就不加；

  - `always` - 给我加！
  - `never` - 你敢加？

简子们好，我们又见面了。之前第5章，我们一起探讨了cargo的一些常用的基本技能。通过第5章的学习，大家基本能解决日常项目开发中遇到的大多数问题。但实际上，cargo提供给我们所使用的功能不仅限于此。我只想说一个字：cargo很好很强大，而且远比你想象的强大。 本章将深入探讨cargo的一些细节问题，这包括以下几个方面：

- 基于语义化版本的项目版本声明与管理
- cargo的toml描述文件配置字段详细参考

基于语义化版本的项目版本声明与管理

我们在使用toml描述文件对项目进行配置时，经常会遇到项目版本声明及管理的问题，比如：

```
[package]
name = "libevent_sys"
version = "0.1.0"

[dependencies]
libc = "0.2"
```

这里package段落中的version字段的值，以及dependencies段落中的libc字段的值，这些值的写法，都涉及到语义化版本控制的问题。语义化版本控制是用一组简单的规则及条件来约束版本号的配置和增长。这些规则是根据（但不局限于）已经被各种封闭、开放源码软件所广泛使用的惯例所设计。简单来说，语义化版本控制遵循下面这些规则：

- 版本格式：主版本号.次版本号.修订号，版本号递增规则如下：
- 主版本号：当你做了不兼容的 API 修改，
- 次版本号：当你做了向下兼容的功能性新增，
- 修订号：当你做了向下兼容的问题修正。
- 先行版本号及版本编译信息可以加到“主版本号.次版本号.修订号”的后面，作为延伸。

关于语义化版本控制的具体细节问题，大家可以参考这里，我不再赘述。

cargo的toml描述文件配置字段详细参考

[package]段落

啥也不多说了，直接上例子，大家注意我在例子中的中文解释，个人觉得这样比较一目了然：

```
[package]
 # 软件包名称，如果需要在别的地方引用此软件包，请用此名称。
name = "hello_world"

# 当前版本号，这里遵循semver标准，也就是语义化版本控制标准。
version = "0.1.0"    # the current version, obeying semver

# 软件所有作者列表
authors = ["you@example.com"]

# 非常有用的一个字段，如果要自定义自己的构建工作流，
# 尤其是要调用外部工具来构建其他本地语言（C、C++、D等）开发的软件包时。
# 这时，自定义的构建流程可以使用rust语言，写在"build.rs"文件中。
build = "build.rs"

# 显式声明软件包文件夹内哪些文件被排除在项目的构建流程之外，
# 哪些文件包含在项目的构建流程中
exclude = ["build/**/*.o", "doc/**/*.html"]
include = ["src/**/*", "Cargo.toml"]

# 当软件包在向公共仓库发布时出现错误时，使能此字段可以阻止此错误。
publish = false

# 关于软件包的一个简短介绍。
description = "..."

# 下面这些字段标明了软件包仓库的更多信息
documentation = "..."
homepage = "..."
repository = "..."
```

```
# 顾名思义，此字段指向的文件就是传说中的ReadMe，
# 并且，此文件的内容最终会保存在注册表数据库中。
readme = "..."

# 用于分类和检索的关键词。
keywords = ["...", "..."]

# 软件包的许可证，必须是cargo仓库已列出的已知的标准许可证。
license = "..."

# 软件包的非标许可证书对应的文件路径。
license-file = "..."
```

依赖的详细配置

最直接的方式在之前第五章探讨过，这里不在赘述，例如这样：

```
[dependencies]
hammer = "0.5.0"
color = "> 0.6.0, < 0.8.0"
```

与平台相关的依赖定义格式不变，不同的是需要定义在[target]字段下。例如：

```
# 注意，此处的cfg可以使用not、any、all等操作符任意组合键值对。
# 并且此用法仅支持cargo 0.9.0（rust 1.8.0）以上版本。
# 如果是windows平台，则需要此依赖。
[target.'cfg(windows)'.dependencies]
winhttp = "0.4.0"

[target.'cfg(unix)'.dependencies]
openssl = "1.0.1"

#如果是32位平台，则需要此依赖。
[target.'cfg(target_pointer_width = "32")'.dependencies]
native = { path = "native/i686" }

[target.'cfg(target_pointer_width = "64")'.dependencies]
native = { path = "native/i686" }

# 另一种写法就是列出平台的全称描述
[target.x86_64-pc-windows-gnu.dependencies]
winhttp = "0.4.0"
[target.i686-unknown-linux-gnu.dependencies]
openssl = "1.0.1"

# 如果使用自定义平台，请将自定义平台文件的完整路径用双引号包含
[target."x86_64/windows.json".dependencies]
winhttp = "0.4.0"
[target."i686/linux.json".dependencies]
openssl = "1.0.1"
native = { path = "native/i686" }
openssl = "1.0.1"
native = { path = "native/x86_64" }

# [dev-dependencies]段落的格式等同于[dependencies]段落，
# 不同之处在于，[dependencies]段落声明的依赖用于构建软件包，
# 而[dev-dependencies]段落声明的依赖仅用于构建测试和性能评估。
# 此外，[dev-dependencies]段落声明的依赖不会传递给其他依赖本软件包的项目
[dev-dependencies]
iron = "0.2"
```

自定义编译器调用方式模板详细参数

cargo内置五种编译器调用模板，分别为dev、release、test、bench、doc，分别用于定义不同类型生成目标时的编译器参数，如果我们自己想改变这些编译模板，可以自己定义相应字段的值，例如（注意：下述例子中列出的值均为此模板字段对应的系统默认值）：

```
# 开发模板, 对应`cargo build`命令
[profile.dev]
opt-level = 0  # 控制编译器的 --opt-level 参数，也就是优化参数
debug = true   # 控制编译器是否开启 `-g` 参数
rpath = false  # 控制编译器的 `-C rpath` 参数
lto = false    # 控制`-C lto` 参数，此参数影响可执行文件和静态库的生成，
debug-assertions = true  # 控制调试断言是否开启
codegen-units = 1 # 控制编译器的 `-C codegen-units` 参数。注意，当`l
to = true`时，此字段值被忽略

# 发布模板, 对应`cargo build --release`命令
[profile.release]
opt-level = 3
debug = false
rpath = false
lto = false
debug-assertions = false
codegen-units = 1

# 测试模板，对应`cargo test`命令
[profile.test]
opt-level = 0
debug = true
rpath = false
lto = false
debug-assertions = true
codegen-units = 1

# 性能评估模板，对应`cargo bench`命令
[profile.bench]
opt-level = 3
debug = false
```

```
rpath = false
lto = false
debug-assertions = false
codegen-units = 1

# 文档模板，对应`cargo doc`命令
[profile.doc]
opt-level = 0
debug = true
rpath = false
lto = false
debug-assertions = true
codegen-units = 1
```

需要注意的是，当调用编译器时，只有位于调用最顶层的软件包的模板文件有效，其他的子软件包或者依赖软件包的模板定义将被顶层软件包的模板覆盖。

[features]段落

[features]段落中的字段被用于条件编译选项或者是可选依赖。例如：

```
[package]
name = "awesome"

[features]
# 此字段设置了可选依赖的默认选择列表，
# 注意这里的"session"并非一个软件包名称，
# 而是另一个featrue字段session
default = ["jquery", "uglifier", "session"]

# 类似这样的值为空的feature一般用于条件编译，
# 类似于`#[cfg(feature = "go-faster")]`。
go-faster = []

# 此feature依赖于bcrypt软件包，
# 这样封装的好处是未来可以对secure-password此feature增加可选项目。
secure-password = ["bcrypt"]

# 此处的session字段导入了cookie软件包中的feature段落中的session字段
session = ["cookie/session"]

[dependencies]
# 必要的依赖
cookie = "1.2.0"
oauth = "1.1.0"
route-recognizer = "=2.1.0"

# 可选依赖
jquery = { version = "1.0.2", optional = true }
uglifier = { version = "1.5.3", optional = true }
bcrypt = { version = "*", optional = true }
civet = { version = "*", optional = true }
```

如果其他软件包要依赖使用上述awesome软件包，可以在其描述文件中这样写：

```
[dependencies.awesome]
version = "1.3.5"
default-features = false # 禁用awesome 的默认features
features = ["secure-password", "civet"] # 使用此处列举的各项feature
s
```

使用features时需要遵循以下规则：

- feature名称在本描述文件中不能与出现的软件包名称冲突
- 除了default feature，其他所有的features均是可选的
- features不能相互循环包含
- 开发依赖包不能包含在内
- features组只能依赖于可选软件包

features的一个重要用途就是，当开发者需要对软件包进行最终的发布时，在进行构建时可以声明暴露给终端用户的features，这可以通过下述命令实现：

```
$ cargo build --release --features "shumway pdf"
```

关于测试

当运行cargo test命令时，cargo将会按做以下事情：

- 编译并运行软件包源代码中被#[cfg(test)] 所标志的单元测试
- 编译并运行文档测试
- 编译并运行集成测试
- 编译examples

配置构建目标

所有的诸如[[bin]], [lib], [[bench]], [[test]]以及 [[example]]等字段，均提供了类似的配置，以说明构建目标应该怎样被构建。例如（下述例子中[lib]段落中各字段值均为默认值）：

```
[lib]
# 库名称，默认与项目名称相同
name = "foo"

# 此选项仅用于[lib]段落，其决定构建目标的构建方式，
# 可以取dylib, rlib, staticlib 三种值之一，表示生成动态库、r库或者静态库。

crate-type = ["dylib"]

# path字段声明了此构建目标相对于cargo.toml文件的相对路径
path = "src/lib.rs"

# 单元测试开关选项
test = true

# 文档测试开关选项
doctest = true

# 性能评估开关选项
bench = true

# 文档生成开关选项
doc = true

# 是否构建为编译器插件的开关选项
plugin = false

# 如果设置为false，`cargo test`将会忽略传递给rustc的--test参数。
harness = true
```

测试与评测

本章讲解 Rust 中内建的测试与评测相关知识。

测试

> 程序测试是一种找到缺陷的有效方式，但是它对证明没有缺陷却无能为力。
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)

作为软件工程质量保障体系的重要一环，测试是应该引起我们充分注意并重视的事情。前面说过，Rust 语言的设计集成了最近十多年中总结出来的大量最佳工程实践，而对测试的原生集成也正体现了这一点。下面来看 Rust 是怎么设计测试特性的。

Rust 的测试特性按精细度划分，分为 3 个层次：

1. 函数级；
2. 模块级；
3. 工程级；

另外，Rust 还支持对文档进行测试。

函数级测试

在本章中，我们用创建一个库的实操来讲解测试的内容。我们先用 cargo 建立一个库工程：`adder`

```
$ cargo new adder
$ cd adder
```

`#[test]` 标识

打开 `src/lib.rs` 文件，可以看到如下代码

```
#[test]
fn it_works() {
    // do test work
}
```

Rust 中，只需要在一个函数的上面，加上 `#[test]` 就标明这是一个测试用的函数。

有了这个属性之后，在使用 `cargo build` 编译时，就会忽略这些函数。使用 `cargo test` 可以运行这些函数。类似于如下效果：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust 提供了两个宏来执行测试断言：

```
assert!(expr)               测试表达式是否为 true 或 false
assert_eq!(expr, expr)      测试两个表达式的结果是否相等
```

比如

```
#[test]
fn it_works() {
    assert!(false);
}
```

运行 `cargo test` ，你会得到类似下面这样的提示

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false',
 /home/steve/tmp/adder/src/lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /home/steve/src
/rust/src/libtest/lib.rs:247
```

`#[should_panic]` 标识

如果你的测试函数没完成，或没有更新，或是故意让它崩溃，但为了让测试能够顺利完成，我们主动可以给测试函数加上 `#[should_panic]` 标识，就不会让 `cargo test` 报错了。

如

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

运行 `cargo test` ，结果类似如下：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`#[ignore]` 标识

有时候，某个测试函数非常耗时，或暂时没更新，我们想不让它参与测试，但是又不想删除它，这时， `#[ignore]` 就派上用场了。

```
#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

写上这个，运行 `cargo test` 的时候，就不会测试这个函数。

模块级测试

有时，我们会组织一批测试用例，这时，模块化的组织结构就有助于建立结构性的测试体系。Rust 中，可以类似如下写法：

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

也即在 `mod` 的上面写上 `#[cfg(test)]` ，表明这个模块是个测试模块。一个测试模块中，可以包含若干测试函数，测试模块中还可以继续包含测试模块，即模块的嵌套。

如此，就形式了结构化的测试体系，甚是方便。

工程级测试

函数级和模块级的测试，代码是与要测试的模块（编译单元）写在相同的文件中，一般做的是白盒测试。工程级的测试，一般做的就是黑盒集成测试了。

我们看一个工程的目录，在这个目录下，有一个 `tests` 文件夹（没有的话，就手动建立）

```
Cargo.toml
Cargo.lock
examples
src
tests
```

我们在 tests 目录下，建立一个文件 `testit.rs` ，名字随便取皆可。内容为：

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

这里，比如，我们 src 中，写了一个库，提供了一个 `add_two` 函数，现在进行集成测试。

首先，用 `extern crate` 的方式，引入这个库，由于是同一个项目，cargo 会自动找。引入后，就按模块的使用方法调用就行了，其它的测试标识与前面相同。

写完后，运行一下 `cargo test` ，提示类似如下：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

文档级测试

Rust 对文档的哲学，是不要单独写文档，一是代码本身是文档，二是代码的注释就是文档。Rust 不但可以自动抽取代码中的文档，形成标准形式的文档集合，还可以对文档中的示例代码进行测试。

比如，我们给上面库加点文档：

```
//! The `adder` crate provides functions that add numbers to oth
er numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

运行 `cargo test`，结果如下：

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

看到了吧，多了些测试结果。

结语

我们可以看到，Rust 对测试，对文档，对文档中的示例代码测试，都有特性支持。从这些细节之处，可以看出 Rust 设计的周密性和严谨性。

但是，光有好工具是不够的，工程的质量更重要的是写代码的人决定的。我们应该在 Rust 严谨之风的熏陶下，养成良好的编码和编写测试的习惯，掌握一定的分析方法，把质量要求贯彻到底。

性能测试

单元测试是用来校验程序的正确性的，然而，程序能正常运行后，往往还需要测试程序（一部分）的执行速度，这时，f就需要用到性能测试。 通常来讲，所谓性能测试，指的是测量程序运行的速度，即运行一次要多少时间（通常是执行多次求平均值）。Rust 竟然连这个特性都集成在语言基础特性中，真的是一门很重视工程性的语言。

下面直接说明如何使用。

```
cargo new benchit
cd benchit
```

编辑 `src/lib.rs` 文件，在里面添加如下代码：

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

注意：

1. 这里虽然使用了 `extern crate test;`，但是项目的 `Cargo.toml` 文件中依赖区并不需要添加对 `test` 的依赖；
2. 评测函数 `fn bench_add_two(b: &mut Bencher) {}` 上面使用 `#[bench]` 做标注，同时函数接受一个参数，`b` 就是 Rust 提供的评测器。这个写法是固定的。

然后，在工程根目录下，执行

```
cargo bench
```

输出结果类似如下：

```
$ cargo bench
   Compiling benchit v0.0.1 (file:///home/mike/tmp/benchit)
     Running target/release/benchit-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

可以看到，Rust 的性能测试是以纳秒 ns 为单位。

写测评代码的时候，需要注意以下一些点：

1. 只把你需要做性能测试的代码（函数）放在评测函数中；
2. 对于参与做性能测试的代码（函数），要求每次测试做同样的事情，不要做累积和改变外部状态的操作；
3. 参数性能测试的代码（函数），执行时间不要太长。太长的话，最好分成几个部分测试。这也方便找出性能瓶颈所在地方。

代码风格

空白

- 每行不能超出99个字符。
- 缩进只用空格，不用TAB。
- 行和文件末尾不要有空白。

空格

- 二元运算符左右加空格，包括属性里的等号：

```
#[deprecated = "Use `bar` instead."]
fn foo(a: usize, b: usize) -> usize {
    a + b
}
```

- 在分号和逗号后面加空格：

```
fn foo(a: Bar);

MyStruct { foo: 3, bar: 4 }

foo(bar, baz);
```

- 在单行语句块或 `struct` 表达式的开始大括号之后和结束大括号之前加空格：

```
spawn(proc() { do_something(); })

Point { x: 0.1, y: 0.3 }
```

折行

- 对于多行的函数签名，每个新行和第一个参数对齐。允许每行多个参数：

```
fn frobnicate(a: Bar, b: Bar,
              c: Bar, d: Bar)
              -> Bar {
    ...
}

fn foo<T: This,
       U: That>(
       a: Bar,
       b: Bar)
       -> Baz {
    ...
}
```

- 多行函数调用一般遵循和签名统一的规则。然而，如果最后的参数开始了一个语句块，块的内容可以开始一个新行，缩进一层：

```
fn foo_bar(a: Bar, b: Bar,
           c: |Bar|) -> Bar {
    ...
}

// 可以在同一行：
foo_bar(x, y, |z| { z.transpose(y) });

// 也可以在新一行缩进函数体：
foo_bar(x, y, |z| {
    z.quux();
    z.rotate(x)
})
```

对齐

常见代码不必在行中用多余的空格来对齐。

```
// 好
struct Foo {
    short: f64,
    really_long: f64,
}

// 坏
struct Bar {
    short:       f64,
    really_long: f64,
}

// 好
let a = 0;
let radius = 7;

// 坏
let b        = 0;
let diameter = 7;
```

避免块注释

使用行注释：

```
// 等待主线程返回，并设置过程错误码
// 明显地。
```

而不是：

```
/*
 * 等待主线程返回，并设置过程错误码
 * 明显地。
 */
```

文档注释

文档注释前面加三个斜线(`///`)而且提示你希望将注释包含在 Rustdoc 的输出里。 它们支持 Markdown 语言 而且是注释你的公开API的主要方式。

支持的 markdown 功能包括列在 GitHub Flavored Markdown 文档中的所有扩展，加上上角标。

总结行

任何文档注释中的第一行应该是一行总结代码的单行短句。该行用于在 Rustdoc 输出中的一个简短的总结性描述，所以，让它短比较好。

句子结构

所有的文档注释，包括总结行，一个以大写字母开始，以句号、问号，或者感叹号结束。最好使用完整的句子而不是片段。

总结行应该以 第三人称单数陈述句形式 来写。 基本上，这意味着用 "Returns" 而不是 "Return"。

例如：

```
/// 根据编译器提供的参数，设置一个缺省的运行时配置。
///
/// 这个函数将阻塞直到整个 M:N 调度器池退出了。
/// 这个函数也要求一个本地的线程可用。
///
/// # 参数
///
/// * `argc` 和 `argv` - 参数向量。在 Unix 系统上，该信息被`os::args`
使用。
///
/// * `main` - 运行在 M:N 调度器池内的初始过程。
///            一旦这个过程退出，调度池将开始关闭。
///            整个池（和这个函数）将只有在所有子线程完成执行后。
///
/// # 返回值
///
/// 返回值被用作进程返回码。成功是 0，101 是错误。
```

避免文档内注释

内嵌文档注释 只用于 注释 crates 和文件级的模块：

```
//! 核心库。
//!
//! 核心库是...
```

解释上下文

Rust 没有特定的构造器，只有返回新实例的函数。 这些在自动生成的类型文档中是不可见的，因此你应该专门链接到它们：

```
/// An iterator that yields `None` forever after the underlying
iterator
/// yields `None` once.
///
/// These can be created through
/// [`iter.fuse()`](trait.Iterator.html#method.fuse).
pub struct Fuse<I> {
    // ...
}
```

开始的大括号总是出现的同一行。

```
fn foo() {
    ...
}

fn frobnicate(a: Bar, b: Bar,
              c: Bar, d: Bar)
              -> Bar {
    ...
}

trait Bar {
    fn baz(&self);
}

impl Bar for Baz {
    fn baz(&self) {
        ...
    }
}

frob(|x| {
    x.transpose()
})
```

`match` 分支有大括号，除非是单行表达式。

```
match foo {
    bar => baz,
    quux => {
        do_something();
        do_something_else()
    }
}
```

`return` 语句有分号。

```
fn foo() {
    do_something();

    if condition() {
        return;
    }

    do_something_else();
}
```

行尾的逗号

```
Foo { bar: 0, baz: 1 }

Foo {
    bar: 0,
    baz: 1,
}

match a_thing {
    None => 0,
    Some(x) => 1,
}
```

一般命名约定

通常，Rust 倾向于为“类型级”结构(类型和 traits)使用 `CamelCase` 而为“值级”结构使用 `snake_case` 。更确切的约定：

条目	约定
Crates	`snake_case` (但倾向于单个词)
Modules	`snake_case`
Types	`CamelCase`
Traits	`CamelCase`
Enum variants	`CamelCase`
Functions	`snake_case`
Methods	`snake_case`
General constructors	`new` 或 `with_more_details`
Conversion constructors	`from_some_other_type`
Local variables	`snake_case`
Static variables	`SCREAMING_SNAKE_CASE`
Constant variables	`SCREAMING_SNAKE_CASE`
Type parameters	简洁 `CamelCase` ，通常单个大写字母： `T`
Lifetimes	短的小写: `'a`

在 `CamelCase` 中, 首字母缩略词被当成一个单词：用 `Uuid` 而不是 `UUID` 。在 `snake_case` 中，首字母缩略词全部是小写： `is_xid_start` 。

在 `snake_case` 或 `SCREAMING_SNAKE_CASE` 中，“单词”永远不应该只包含一个字母， 除非是最后一个“单词”。所以，我们有 `btree_map` 而不是 `b_tree_map` ， `PI_2` 而不是 `PI2` 。

引用函数/方法名中的类型

函数名经常涉及类型名，最常见的约定例子像 `as_slice` 。如果类型有一个纯粹的文本名字（忽略参数）， 在类型约定和函数约定之间转换是直截了当的：

类型名	方法中的文本
`String`	`string`
`Vec<T>`	`vec`
`YourType`	`your_type`

涉及记号的类型遵循以下约定。这些规则有重叠；应用最适用的规则：

类型名	方法中的文本
`&str`	`str`
`&[T]`	`slice`
`&mut [T]`	`mut_slice`
`&[u8]`	`bytes`
`&T`	`ref`
`&mut T`	`mut`
`*const T`	`ptr`
`*mut T`	`mut_ptr`

避免冗余的前缀

一个模块中的条目的名字不应拿模块的名字做前缀：

倾向于

```
mod foo {
    pub struct Error { ... }
}
```

而不是

```
mod foo {
    pub struct FooError { ... }
}
```

这个约定避免了口吃（像 `io::IoError` ）。库客户端可以在导入时重命名以避免冲突。

Getter/setter 方法

一些数据类型不希望提供对它们的域的直接访问，但是提供了 "getter" 和 "setter" 方法用于操纵域状态 （经常提供检查或其他功能）。

域 `foo: T` 的约定是：

- 方法 `foo(&self) -> &T` 用于获得该域的当前值。
- 方法 `set_foo(&self, val: T)` 用于设置域。（这里的 `val` 参数可能取 `&T` 或其他类型，取决于上下文。）

请注意，这个约定是关于通常数据类型的 getters/setters， 不是 关于构建者对象的。

断言

- 简单的布尔断言应该加上 `is_` 或者其他的简短问题单词作为前缀，e.g.，`is_empty` 。
- 常见的例外： `lt` ， `gt` ，和其他已经确认的断言名。

导入

一个 crate/模块的导入应该按顺序包括下面各个部分，之间以空行分隔：

- `extern crate` 指令
- 外部 `use` 导入
- 本地 `use` 导入
- `pub use` 导入

例如：

```
// Crates.
extern crate getopts;
extern crate mylib;

// 标准库导入。
use getopts::{optopt, getopts};
use std::os;

// 从一个我们写的库导入。
use mylib::webserver;

// 当我们导入这个模块时会被重新导出。
pub use self::types::Webdata;
```

避免 `use *`，除非在测试里

Glob 导入有几个缺点：

- 更难知道名字在哪里绑定。
- 它们前向不兼容，因为新的上流导出可能与现存的名字冲突。

在写 `test` 子模块时，为方便导入 `super::*` 是合适的。

当模块限定函数时，倾向于完全导入类型/traits。

例如：

```
use option::Option;
use mem;

let i: isize = mem::transmute(Option(0));
```

在 crate 级重新导出最重要的类型。

Crates `pub use` 最常见的类型为方便，因此，客户端不必记住或写 crate 的模块结构以使用这些类型。

类型和操作在一起定义。

类型定义和使用它们的函数/模块应该在同一模块中定义，类型出现在函数/模块前面。

Any和反射

熟悉Java的同学肯定对Java的反射能力记忆犹新，同样的，Rust也提供了运行时反射的能力。但是，这里有点小小的不同，因为 Rust 不带 VM 不带 Runtime ,因此，其提供的反射更像是一种编译时反射。

因为，Rust只能对 `'static` 生命周期的变量（常量）进行反射！

举个例子

我们会有这样的需求，去某些路径里加载配置文件。我们可能提供一个配置文件路径，好吧，这是个字符串(`String`)。但是，当我想要传入多个配置文件的路径的时候怎们办？理所应当的，我们传入了一个数组。

这下可坏了……Rust不支持重载啊！于是有人就很单纯的写了两个函数～～！

其实不用……我们只需要这么写……

```
use std::any::Any;
use std::fmt::Debug ;

fn load_config<T:Any+Debug>(value: &T) -> Vec<String>{
    let mut cfgs: Vec<String>= vec![];
    let value = value as &Any;
    match value.downcast_ref::<String>() {
        Some(cfp) => cfgs.push(cfp.clone()),
        None => (),
    };

    match value.downcast_ref::<Vec<String>>() {
        Some(v) => cfgs.extend_from_slice(&v),
        None =>(),
    }

    if cfgs.len() == 0 {
        panic!("No Config File");
    }
    cfgs
}

fn main() {
    let cfp = "/etc/wayslog.conf".to_string();
    assert_eq!(load_config(&cfp), vec!["/etc/wayslog.conf".to_st
ring()]);
    let cfps = vec!["/etc/wayslog.conf".to_string(),
                    "/etc/wayslog_sec.conf".to_string()];
    assert_eq!(load_config(&cfps),
               vec!["/etc/wayslog.conf".to_string(),
                    "/etc/wayslog_sec.conf".to_string()]);
}
```

我们来重点分析一下中间这个函数：

```
fn load_config<T:Any+Debug>(value: &T) -> Vec<String>{..}
```

首先，这个函数接收一个泛型 `T` 类型，`T` 必须实现了 `Any` 和 `Debug` 。

这里可能有同学疑问了，你不是说只能反射 `'static` 生命周期的变量么？我们来看一下 `Any` 限制：

```
pub trait Any: 'static + Reflect {
    fn get_type_id(&self) -> TypeId;
}
```

看， `Any` 在定义的时候就规定了其生命周期，而 `Reflect` 是一个Marker，默认所有的Rust类型都会实现他！注意，这里不是所有原生类型，而是所有类型。

好的，继续，由于我们无法判断出传入的参数类型，因此，只能从运行时候反射类型。

```
let value = value as &Any;
```

首先，我们需要将传入的类型转化成一个 `trait Object` , 当然了，你高兴的话用 `UFCS` 也是可以做的，参照本章最后的附录。

这样，value 就可以被堪称一个 Any 了。然后，我们通过 `downcast_ref` 来进行类型推断。如果类型推断成功，则 value 就会被转换成原来的类型。

有的同学看到这里有点懵，为什么你都转换成 Any 了还要转回来？

其实，转换成 Any 是为了有机会获取到他的类型信息，转换回来，则是为了去使用这个值本身。

最后，我们对不同的类型处以不同的处理逻辑。最终，一个反射函数就完成了。

说说注意的地方

需要注意的是，Rust本身提供的反射能力并不是很强大。相对而言只能作为一个辅助的手段。并且，其只能对 `'static` 周期进行反射的限制，的确限制了其发挥。还有一点需要注意的是，Rust的反射只能被用作类型推断，绝对不能被用作接口断言！

啥，你问原因？因为写不出来啊……

安全（Safety）

本章不讲解任何语言知识点，而是对 Rust 安全理念的一些总结性说明。

安全，本身是一个相当大的话题。安全性，本身也需要一个局部性的定义。

Rust 的定义中，凡是 可能 会导致程序内存使用出错的特性，都被认为是 不安全的（**unsafe**）。反之，则是 安全的（**safe**）。

基于这种定义，C 语言，基本是不安全的语言（它是众多不安全特性的集合。特别是指针相关特性，多线程相关特性）。

Rust 的这个定义，隐含了一个先决假设：人之初，性本恶。人是不可靠的，人是会犯错误的，即 Rust 不相信人的实施过程。在这一点上，C 语言的理念与之完全相反：C 语言完全相信人，人之初，性本善，由人进行完全地控制。

根据 Rust 的定义，C 语言几乎是不安全的代名字。但是，从本质上来说，一段程序是否安全，并不由开发它的语言决定。用 C 语言开发出的程序，不一定就是不安全的代码，只不过相对来说，需要花更多的精力进行良好的设计和长期的实际运行验证。Rust 使开发出安全可靠的代码相对容易了。

世界本身是肮脏的。正如，纯函数式语言中还必须有用于处理副作用的 `Monad` 存在一样，Rust 仅凭安全的特性集合，也是无法处理世界的所有结构和问题的。所以，Rust 中，还有 `unsafe` 部分的存在。实际上，Rust 的 std 本身也是建立在大量 `unsafe` 代码的基础之上的。所以，世界就是纯粹建立在不纯粹之上，“安全”建立在“不安全”之上。

因此，Rust 本身可以被认为是两种编程语言的混合：`Safe Rust` 和 `Unsafe Rust` 。

只使用 `Safe Rust` 的情况下，你不需要担心任何类型安全性和内存安全性的问题。你永远不用忍受空指针，悬挂指针或其它可能的未定义行为的干扰。

`Unsafe Rust` 在 `Safe Rust` 的所有特性上，只给程序员开放了以下四种能力：

1. 对原始指针进行解引（Dereference raw pointers）；
2. 调用 `unsafe` 函数（包括 C 函数，内部函数，和原始分配器）；
3. 实现 `unsafe` traits；

4. 修改（全局）静态变量。

上述这四种能力，如果误用的话，会导致一些未定义行为，具有不确定后果，很容易引起程序崩溃。

Rust 中定义的不确定性行为有如下一些：

1. 对空指针或悬挂指针进行解引用；
2. 读取未初始化的内存；
3. 破坏指针重命名规则（比如同一资源的 `&mut` 引用不能出现多次，`&mut` 与 `&` 不能同时出现）；
4. 产生无效的原生值：
    - 空指针，悬挂指针；
    - bool 值不是 0 或 1；
    - 未定义的枚举取值；
    - char 值超出取值范围 [0x0, 0xD7FF] 和 [0xE000, 0x10FFFF]；
    - 非 utf-8 字符串；
5. Unwinding 到其它语言中；
6. 产生一个数据竞争。

以下一些情况，Rust 认为不属于安全性的处理范畴，即认为它们是“安全”的：

1. 死锁；
2. 存在竞争条件；
3. 内存泄漏；
4. 调用析构函数失败；
5. 整数溢出；
6. 程序被中断；
7. 删除产品数据库（:D）；

参考

下面一些链接，给出了安全性更详细的讲解（部分未来会有对应的中文翻译）。

- Unsafe
- Meet Safe and Unsafe
- How Safe and Unsafe Interact
- 蓦然回首万事空 ————空指针漫谈

常用数据结构实现

本章讲解如何使用 Rust 进行一些常用数据结构的实现。实现的代码仅作示例用，并不一定十分高效。真正使用的时候，请使用标准库或第三方成熟库中的数据结构。

栈

栈简介

- 栈作为一种数据结构，是一种只能在一端进行插入和删除操作的特殊线性表。
- 它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。

> 栈（stack）又名堆栈，它是一种运算受限的线性表。其限制是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。

栈的实现步骤：

- 定义一个栈结构 `Stack`
- 定义组成栈结构的栈点 `StackNode`
- 实现栈的初始化函数 `new( )`
- 实现进栈函数 `push( )`
- 实现退栈函数 `pop( )`

定义一个栈结构 `Stack`

```
#[derive(Debug)]
struct Stack<T> {
    top: Option<Box<StackNode<T>>>,
}
```

让我们一步步来分析

- 第一行的 `#[derive(Debug)]` 是为了让 `Stack` 结构体可以打印调试。

- 第二行是定义了一个 `Stack` 结构体，这个结构体包含一个泛型参数 `T` 。
- 第三行比较复杂，在定义 `StackNode` 的时候介绍

定义组成栈结构的栈点 `StackNode`

```
#[derive(Clone,Debug)]
struct StackNode<T> {
    val: T,
    next: Option<Box<StackNode<T>>>,
}
```

在这段代码的第三行， 我们定义了一个 `val` 保存 `StackNode` 的值。

> 现在我们重点来看看第四行： 我们从里到外拆分来看看，首先是 `Box<StackNode<T>>` ，这里的 `Box` 是 Rust 用来显式分配堆内存的类型：
>
> `pub struct Box<T> where T: ?Sized(_);`
> 详细文档请参考Rust的标准库
>
> 在 Rust 里面用强大的类型系统做了统一的抽象。在这里相当于在堆空间里申请了一块内存保存 `StackNode<T>` 。
>
> 为什么要这么做了？如果不用**Box**封装会怎么样呢？
>
> 如果不用 Box 封装，rustc 编译器会报错，在 Rust 里面，rustc 默认使用栈空间，但是这里的 `StackNode` 定义的时候使用了递归的数据结构，next 属性的类型是 `StackNode<T>` ，而这个类型是无法确定大小的，所有这种无法确定大小的类型，都不能保存在栈空间。所以需要使用 `Box` 来封装。这样的话 `next` 的类型就是一个指向某一块堆空间的指针，而指针是可以确定大小的，因此能够保存在栈空间。
>
> 那么为什么还需要使用 **`Option`** 来封装呢？
>
> `Option` 是 Rust 里面的一个抽象类型，定义如下：

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Option 里面包括元素，None 和 Some(T) ，这样就很轻松的描述了 next 指向栈尾的元素的时候，都是在 Option 类型下，方便了功能实现，也方便了错误处理。Option 还有很多强大的功能，读者可以参考下面几个连接：

Option标准库文档

Error Handling in Rust

rustbyexample 的 Options with Results部分

实现 `new( ) push( ) pop( )`

接下来是实现 stack 的主要功能了。

```
impl<T> Stack<T> {
    fn new() -> Stack<T> {
        Stack{ top: None }
    }

    fn push(&mut self, val: T) {
        let mut node = StackNode::new(val);
        let next = self.top.take();
        node.next = next;
        self.top = Some(Box::new(node));
    }

    fn pop(&mut self) -> Option<T> {
        let val = self.top.take();
        match val {
            None => None,
            Some(mut x) => {
                self.top = x.next.take();
                Some(x.val)
            },
        }
    }
}
```

- `new( )` 比较简单，Stack 初始化的时候为空，栈顶元素 `top` 就没有任何值，所以 `top` 为 `None` 。
- `push( )` 的主要功能是往栈里面推入元素，把新的 StackNode 指向 Stack 里面旧的值，同时更新 Stack 栈顶指向新进来的值。

> 这里有个需要注意的地方是第8行代码里面， `let next = self.top.take();` ，使用了 Option 类型的 take 方法： `fn take(&mut self) -> Option<T>` 它会把 Option 类型的值取走，并把它的元素改为 None

- `pop( )` 的功能是取出栈顶的元素，如果栈顶为 None 则返回 None。

完整代码（包含简单的测试）

```
#[derive(Debug)]
struct Stack<T> {
    top: Option<Box<StackNode<T>>>,
}

#[derive(Clone,Debug)]
struct StackNode<T> {
    val: T,
    next: Option<Box<StackNode<T>>>,
}

impl <T> StackNode<T> {
    fn new(val: T) -> StackNode<T> {
        StackNode { val: val, next: None }
    }
}

impl<T> Stack<T> {
    fn new() -> Stack<T> {
        Stack{ top: None }
    }

    fn push(&mut self, val: T) {
        let mut node = StackNode::new(val);
        let next = self.top.take();
        node.next = next;
        self.top = Some(Box::new(node));
    }

    fn pop(&mut self) -> Option<T> {
        let val = self.top.take();
        match val {
            None => None,
            Some(mut x) => {
                self.top = x.next.take();
                Some(x.val)
            },
        }
    }
```

```
}

fn main() {
    #[derive(PartialEq,Eq,Debug)]
    struct TestStruct {
        a: i32,
    }

    let a = TestStruct{ a: 5 };
    let b = TestStruct{ a: 9 };

    let mut s = Stack::<&TestStruct>::new();
    assert_eq!(s.pop(), None);

    s.push(&a);
    s.push(&b);
    println!("{:?}", s);

    assert_eq!(s.pop(), Some(&b));
    assert_eq!(s.pop(), Some(&a));
    assert_eq!(s.pop(), None);
}
```

队列

队列简介

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

> 在队列的形成过程中，可以利用线性链表的原理，来生成一个队列。基于链表的队列，要动态创建和删除节点，效率较低，但是可以动态增长。队列采用的**FIFO(first in first out)**，新元素（等待进入队列的元素）总是被插入到链表的尾部，而读取的时候总是从链表的头部开始读取。每次读取一个元素，释放一个元素。所谓的动态创建，动态释放。因而也不存在溢出等问题。由于链表由结构体间接而成，遍历也方便。

队列实现

下面看一下我们使用 Vec 来实现的简单 Queue：

主要实现的 `new( ), push( ), pop( )` 三个方法

```
#[derive(Debug)]
struct Queue<T> {
    qdata: Vec<T>,
}

impl <T> Queue<T> {
    fn new() -> Self {
        Queue{qdata: Vec::new()}
    }

    fn push(&mut self, item:T) {
        self.qdata.push(item);
    }

    fn pop(&mut self) -> T{
        self.qdata.remove(0)
    }
}

fn main() {
    let mut q = Queue::new();
    q.push(1);
    q.push(2);
    println!("{:?}", q);
    q.pop();
    println!("{:?}", q);
    q.pop();
}
```

练习

看起来比我们在上一节实现的Stack简单多了。不过这个Queue实现是有Bug的。

练习：在这个代码的上找到 Bug，并修改。

提示： `pop( )` 方法有 Bug，请参考 Stack 小节的实现，利用 Option 来处理。

二叉树

二叉树简介

在计算机科学中，二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。

> 二叉查找树的子节点与父节点的键一般满足一定的顺序关系，习惯上，左节点的键少于父亲节点的键，右节点的键大于父亲节点的键。
>
> 二叉堆是一种特殊的堆，二叉堆是完全二元树（二叉树）或者是近似完全二元树（二叉树）。二叉堆有两种：最大堆和最小堆。最大堆：父结点的键总是大于或等于任何一个子节点的键；最小堆：父结点的键总是小于或等于任何一个子节点的键。
>
> 二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)，二叉树的子树有左右之分，次序不能颠倒。二叉树的第i层至多有2^{i-1}个结点；深度为k的二叉树至多有2^k-1个结点；对任何一棵二叉树T，如果其终端结点数为n_0，度为2的结点数为n_2，则n_0=n_2+1。
>
> 一棵深度为k，且有2^k-1个节点称之为满二叉树；深度为k，有n个节点的二叉树，当且仅当其每一个节点都与深度为k的满二叉树中，序号为1至n的节点对应时，称之为完全二叉树。

二叉树与树的区别

二叉树不是树的一种特殊情形，尽管其与树有许多相似之处，但树和二叉树有两个主要差别：

1. 树中结点的最大度数没有限制，而二叉树结点的最大度数为2。
2. 树的结点无左、右之分，而二叉树的结点有左、右之分。

定义二叉树的结构

二叉树的每个节点由键key、值value与左右子树left/right组成，这里我们把节点声明为一个泛型结构。

```
type TreeNode<K,V> = Option<Box<Node<K,V>>>;
#[derive(Debug)]
struct Node<K,V: std::fmt::Display> {
   left: TreeNode<K,V>,
   right: TreeNode<K,V>,
   key: K,
   value: V,
}
```

实现二叉树的初始化与二叉查找树的插入

由于二叉查找树要求键可排序，我们要求K实现PartialOrd

```
trait BinaryTree<K,V> {
    fn pre_order(&self);
    fn in_order(&self);
    fn pos_order(&self);
}
trait BinarySearchTree<K:PartialOrd,V>:BinaryTree<K,V> {
    fn insert(&mut self, key:K,value: V);
}
impl<K,V:std::fmt::Display> Node<K,V> {
    fn new(key: K,value: V) -> Self {
        Node{
            left: None,
            right: None,
            value: value,
            key: key,
        }
    }
}
impl<K:PartialOrd,V:std::fmt::Display> BinarySearchTree<K,V> for
 Node<K,V>{
    fn insert(&mut self, key:K,value:V) {
        if self.key < key {
            if let Some(ref mut right) = self.right {
                right.insert(key,value);
            } else {
                self.right = Some(Box::new(Node::new(key,value))
);
            }
        } else {
            if let Some(ref mut left) = self.left {
                left.insert(key,value);
            } else {
                self.left = Some(Box::new(Node::new(key,value)))
;
            }
        }
    }
}
```

二叉树的遍历

- 先序遍历：首先访问根，再先序遍历左（右）子树，最后先序遍历右（左）子树。
- 中序遍历：首先中序遍历左（右）子树，再访问根，最后中序遍历右（左）子树。
- 后序遍历：首先后序遍历左（右）子树，再后序遍历右（左）子树，最后访问根。

下面是代码实现：

```
impl<K,V:std::fmt::Display> BinaryTree<K,V> for Node<K,V> {
    fn pre_order(&self) {
        println!("{}", self.value);

        if let Some(ref left) = self.left {
            left.pre_order();
        }
        if let Some(ref right) = self.right {
            right.pre_order();
        }
    }

    fn in_order(&self) {
        if let Some(ref left) = self.left {
            left.in_order();
        }
        println!("{}", self.value);
        if let Some(ref right) = self.right {
            right.in_order();
        }
    }
    fn pos_order(&self) {
        if let Some(ref left) = self.left {
            left.pos_order();
        }
        if let Some(ref right) = self.right {
            right.pos_order();
        }
        println!("{}", self.value);
    }
}
```

测试代码

```
type BST<K,V> = Node<K,V>;

fn test_insert() {
    let mut root = BST::<i32,i32>::new(3,4);
    root.insert(2,3);
    root.insert(4,6);
    root.insert(5,5);
    root.insert(6,6);
    root.insert(1,8);
    if let Some(ref left) = root.left {
        assert_eq!(left.value, 3);
    }

    if let Some(ref right) = root.right {
        assert_eq!(right.value, 6);
        if let Some(ref right) = right.right {
            assert_eq!(right.value, 5);
        }
    }
    println!("Pre Order traversal");
    root.pre_order();
    println!("In Order traversal");
    root.in_order();
    println!("Pos Order traversal");
    root.pos_order();
}

fn main() {
    test_insert();
}
```

练习

基于以上代码，修改成二叉堆的形式。

优先队列

简介

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （largest-in，first-out）的行为特征。

> 优先队列是0个或多个元素的集合，每个元素都有一个优先权或值，对优先队列执行的操作有：

1. 查找;
2. 插入一个新元素;
3. 删除。

在最小优先队列(min priority queue)中，查找操作用来搜索优先权最小的元素，删除操作用来删除该元素；对于最大优先队列(max priority queue)，查找操作用来搜索优先权最大的元素，删除操作用来删除该元素。优先权队列中的元素可以有相同的优先权，查找与删除操作可根据任意优先权进行。

优先队列的实现：

首先定义 PriorityQueue 结构体

```
#[derive(Debug)]
struct PriorityQueue<T> where T: PartialOrd + Clone {
    pq: Vec<T>
}
```

第二行的 `where T: PartialOrd + Clone` 指的是 PriorityQueue 存储的泛型 T 是满足 `PartialOrd` 和 `Clone` trait 约束的，意味着泛型 T 是可排序和克隆的。

后面是一些基本的方法实现，比较简单，就直接看代码吧。这个优先队列是基于Vec实现的，有O(1)的插入和O(n)的最大/最小值出列。

```
impl<T> PriorityQueue<T> where T: PartialOrd + Clone {
    fn new() -> PriorityQueue<T> {
        PriorityQueue { pq: Vec::new() }
    }

    fn len(&self) -> usize {
        self.pq.len()
    }

    fn is_empty(&self) -> bool {
        self.pq.len() == 0
    }

    fn insert(&mut self, value: T) {
        self.pq.push(value);
    }

    fn max(&self) -> Option<T> {
        if self.is_empty() { return None }
        let max = self.max_index();
        Some(self.pq[max].clone())
    }

    fn min(&self) -> Option<T> {
        if self.is_empty() { return None }
        let min = self.min_index();
        Some(self.pq[min].clone())
    }

    fn delete_max(&mut self) -> Option<T> {
        if self.is_empty() { return None; }
        let max = self.max_index();
        Some(self.pq.remove(max).clone())
    }

    fn delete_min(&mut self) -> Option<T> {
        if self.is_empty() { return None; }
        let min = self.min_index();
        Some(self.pq.remove(min).clone())
```

```
    }

    fn max_index(&self) -> usize {
        let mut max = 0;
        for i in 1..self.pq.len() - 1 {
            if self.pq[max] < self.pq[i] {
                max = i;
            }
        }
        max
    }

    fn min_index(&self) -> usize {
        let mut min = 0;
        for i in 0..self.pq.len() - 1 {
            if self.pq[i] < self.pq[i + 1] {
                min = i;
            }
        }
        min
    }
}
```

测试代码：

```
fn test_keep_min() {
    let mut pq = PriorityQueue::new();
    pq.insert(3);
    pq.insert(2);
    pq.insert(1);
    pq.insert(4);
    assert!(pq.min().unwrap() == 1);
}

fn test_keep_max() {
    let mut pq = PriorityQueue::new();
    pq.insert(2);
    pq.insert(4);
```

```
    pq.insert(1);
    pq.insert(3);
    assert!(pq.max().unwrap() == 4);
}

fn test_is_empty() {
    let mut pq = PriorityQueue::new();
    assert!(pq.is_empty());
    pq.insert(1);
    assert!(!pq.is_empty());
}

fn test_len() {
    let mut pq = PriorityQueue::new();
    assert!(pq.len() == 0);
    pq.insert(2);
    pq.insert(4);
    pq.insert(1);
    assert!(pq.len() == 3);
}

fn test_delete_min() {
    let mut pq = PriorityQueue::new();
    pq.insert(3);
    pq.insert(2);
    pq.insert(1);
    pq.insert(4);
    assert!(pq.len() == 4);
    assert!(pq.delete_min().unwrap() == 1);
    assert!(pq.len() == 3);
}

fn test_delete_max() {
    let mut pq = PriorityQueue::new();
    pq.insert(2);
    pq.insert(10);
    pq.insert(1);
    pq.insert(6);
    pq.insert(3);
    assert!(pq.len() == 5);
```

```
    assert!(pq.delete_max().unwrap() == 10);
    assert!(pq.len() == 4);
}

fn main() {
    test_len();
    test_delete_max();
    test_delete_min();
    test_is_empty();
    test_keep_max();
    test_keep_min();
}
```

练习

基于二叉堆实现一个优先队列，以达到O(1)的出列和O(log n)的入列

链表

链表简介

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 由于不必须按顺序存储，链表在给定位置插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是在有序数据中查找一个节点或者访问特定下标的节点则需要O(n)的时间，而线性表相应的时间复杂度分别是O(logn)和O(1)。

> 使用链表结构可以克服数组需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在内存或磁盘上的顺序，数据的存取往往要在不同的排列顺序中转换。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

下面看我们一步步实现链表：

定义链表结构

```
use List::*;

enum List {
    // Cons: 包含一个元素和一个指向下一个节点的指针的元组结构
    Cons(u32, Box<List>),
    // Nil: 表示一个链表节点的末端
    Nil,
}
```

实现链表的方法

```
impl List {
    // 创建一个空链表
    fn new() -> List {
        // `Nil` 是 `List`类型的。因为前面我们使用了 `use List::*;`
        // 所以不需要 List::Nil 这样使用
        Nil
    }

    // 在前面加一个元素节点，并且链接旧的链表和返回新的链表
    fn prepend(self, elem: u32) -> List {
        // `Cons` 也是 List 类型的
        Cons(elem, Box::new(self))
    }

    // 返回链表的长度
    fn len(&self) -> u32 {
        // `self` 的类型是 `&List`, `*self` 的类型是 `List`,
        // 匹配一个类型 `T` 好过匹配一个引用 `&T`
        match *self {
            // 因为`self`是借用的，所以不能转移 tail 的所有权
            // 因此使用 tail 的引用
            Cons(_, ref tail) => 1 + tail.len(),
            // 基本规则：所以空的链表长度都是0
            Nil => 0
        }
    }

    // 返回连链表的字符串表达形式
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` 和 `print!` 很像
                // 但是返回一个堆上的字符串去替代打印到控制台
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
```

```
    }
}
```

代码测试

```
fn main() {
    let mut list = List::new();

    list = list.prepend(1);
    list = list.prepend(2);
    list = list.prepend(3);

    println!("linked list has length: {}", list.len());
    println!("{}", list.stringify());
}
```

练习

基于以上代码实现一个双向循环链表。

> 双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表。 循环链表是另一种形式的链式存贮结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。

图

图的存储结构

图的存储结构除了要存储图中各个顶点的本身信息外，同时还要存储顶点与顶点之间的所有关系(边的信息)，因此，图的结构比较复杂，很难以数据元素在存储区中的物理位置来表示元素之间的关系，但也正是由于其任意的特性，故物理表示方法很多。常用的图的存储结构有邻接矩阵、邻接表等。

邻接矩阵表示法

对于一个具有n个结点的图，可以使用n*n的矩阵(二维数组)来表示它们间的邻接关系。矩阵 A(i,j) = 1 表示图中存在一条边 (Vi,Vj),而A(i,j)=0表示图中不存在边 (Vi,Vj)。 实际编程时，当图为不带权图时，可以在二维数组中存放 bool 值。

- A(i,j) = true 表示存在边 (Vi,Vj),
- A(i,j) = false 表示不存在边 (Vi,Vj);

当图带权值时，则可以直接在二维数值中存放权值，A(i,j) = null 表示不存在边 (Vi,Vj)。

下面看看我们使用邻接矩阵实现的图结构：

```
#[derive(Debug)]
struct Node {
    nodeid: usize,
    nodename: String,
}

#[derive(Debug,Clone)]
struct Edge {
    edge: bool,
}

#[derive(Debug)]
struct Graphadj {
```

```
    nodenums: usize,
    graphadj: Vec<Vec<Edge>>,
}

impl Node {
    fn new(nodeid: usize, nodename: String) -> Node {
        Node{
            nodeid: nodeid,
            nodename: nodename,
        }
    }
}
impl Edge {
    fn new() -> Edge {
        Edge{
            edge: false,
        }
    }
    fn have_edge() -> Edge {
        Edge{
            edge: true,
        }
    }
}

impl Graphadj {
    fn new(nums:usize) -> Graphadj {
        Graphadj {
            nodenums: nums,
            graphadj: vec![vec![Edge::new();nums]; nums],
        }
    }

    fn insert_edge(&mut self, v1: Node, v2:Node) {
        match v1.nodeid < self.nodenums && v2.nodeid<self.nodenu
ms {
            true => {
                self.graphadj[v1.nodeid][v2.nodeid] = Edge::have
_edge();
                //下面这句注释去掉相当于把图当成无向图
```

```
                //self.graphadj[v2.nodeid][v1.nodeid] = Edge::ha
ve_edge();
            }
            false => {
                panic!("your nodeid is bigger than nodenums!");
            }
        }
    }
}

fn main() {
    let mut g = Graphadj::new(2);
    let v1 = Node::new(0, "v1".to_string());
    let v2 = Node::new(1, "v2".to_string());
    g.insert_edge(v1,v2);
    println!("{:?}", g);
}
```

邻接表表示法

邻接表是图的一种最主要存储结构，用来描述图上的每一个点。

> 实现方式：对图的每个顶点建立一个容器（n个顶点建立n个容器），第i个容器中的结点包含顶点Vi的所有邻接顶点。实际上我们常用的邻接矩阵就是一种未离散化每个点的边集的邻接表。

- 在有向图中，描述每个点向别的节点连的边（点 a->点 b 这种情况）。
- 在无向图中，描述每个点所有的边(点 a->点 b这种情况)

与邻接表相对应的存图方式叫做边集表，这种方法用一个容器存储所有的边。

练习：

实现链接表表示法的图结构。

标准库示例

好了，本书到这里也接近完结了。相信你一在学习了这么多内容的之后，一定跃跃欲试了吧？ 下面，我们将以代码为主，讲解几个利用 `std` 库，即标准库来做的例子。希望大家能从中学到一点写法，并开始自己的Rust之旅。

- 注： 由于笔者的电脑是openSUSE Linux的，所以本章所有代码均只在 `openSUSE Leap 42.1 && rustc 1.9.0-nightly (52e0bda64 2016-03-05)` 下编译通过，对Linux适配可能会更好一点，其他系统的同学请自行参照。

另：本章原本设计的时候附加有时间api的处理，但是在本章写作的时候Rust的大部分时间API还处于Unstable状态，随时可能遭到删除或重写。因此，我们暂时删除了时间API的操作。等以后Rust的API稳定之后，再来补齐这一节。

1. 系统命令:调用grep
2. 目录操作:简单grep
3. 网络模块:W回音

系统命令:调用grep

我们知道，Linux系统中有一个命令叫grep，他能对目标文件进行分析并查找相应字符串，并该字符串所在行输出。 今天，我们先来写一个Rust程序，来调用一下这个 grep 命令

```
use std::process::*;
use std::env::args;

// 实现调用grep命令搜索文件
fn main() {
    let mut arg_iter = args();
    // panic if there is no one
    arg_iter.next().unwrap();
    let pattern = arg_iter.next().unwrap_or("main".to_string());
    let pt =  arg_iter.next().unwrap_or("./".to_string());
    let output = Command::new("/usr/bin/grep")
        .arg("-n")
        .arg("-r")
        .arg(&pattern)
        .arg(&pt)
        .output()
        .unwrap_or_else(|e| panic!("wg panic because:{}", e));
    println!("output:");
    let st = String::from_utf8_lossy(&output.stdout);
    let lines = st.split("\n");
    for line in lines {
        println!("{}", line);
    }
}
```

看起来好像还不错，但是，以上的程序有一个比较致命的缺点——因为Output是同步的，因此，一旦调用的目录下有巨大的文件，grep的分析将占用巨量的时间。这对于一个高可用的程序来说是不被允许的。

那么如何改进呢？

其实在上面的代码中，我们隐藏了一个 `Child` 的概念，即——子进程。

下面我来演示怎么操作子进程：

```
use std::process::*;
use std::env::args;

// 实现调用grep命令搜索文件
fn main() {
    let mut arg_iter = args();
    // panic if there is no one
    arg_iter.next();
    let pattern = arg_iter.next().unwrap_or("main".to_string());
    let pt =  arg_iter.next().unwrap_or("./".to_string());
    let child = Command::new("grep")
        .arg("-n")
        .arg("-r")
        .arg(&pattern)
        .arg(&pt)
        .spawn().unwrap();
    // 做些其他的事情
    std::thread::sleep_ms(1000);
    println!("{}", "计算很费时间……");
    let out = child.wait_with_output().unwrap();
    let out_str = String::from_utf8_lossy(&out.stdout);
    for line in out_str.split("\n") {
        println!("{}", line);
    }
}
```

但是，这个例子和我们预期的并不太一样！

```
./demo main /home/wayslog/rust/demo/src
/home/wayslog/rust/demo/src/main.rs:5:fn main() {
/home/wayslog/rust/demo/src/main.rs:9:    let pattern = arg_iter
.next().unwrap_or("main".to_string());
计算很费时间……
```

为什么呢？

很简单，我们知道，在Linux中， `fork` 出来的函数会继承父进程的所有句柄。因此，子进程也就会继承父进程的标准输出，也就是造成了这样的问题。这也是最后我们用out无法接收到最后的输出也就知道了，因为在前面已经被输出出来了呀！

那么怎么做呢？给这个子进程一个pipeline就好了！

```
use std::process::*;
use std::env::args;

// 实现调用grep命令搜索文件
fn main() {
    let mut arg_iter = args();
    // panic if there is no one
    arg_iter.next();
    let pattern = arg_iter.next().unwrap_or("main".to_string());
    let pt =  arg_iter.next().unwrap_or("./".to_string());
    let child = Command::new("grep")
        .arg("-n")
        .arg("-r")
        .arg(&pattern)
        .arg(&pt)
        // 设置pipeline
        .stdout(Stdio::piped())
        .spawn().unwrap();
    // 做些其他的事情
    std::thread::sleep_ms(1000);
    println!("{}", "计算很费时间……");
    let out = child.wait_with_output().unwrap();
    let out_str = String::from_utf8_lossy(&out.stdout);
    for line in out_str.split("\n") {
        println!("{}", line);
    }
}
```

这段代码相当于给了 `stdout` 一个缓冲区，这个缓冲区直到我们计算完成之后才被读取，因此就不会造成乱序输出的问题了。

这边需要注意的一点是，一旦你开启了一个子进程，那么，无论你程序是怎么处理的，最后一定要记得对这个 `child` 调用 `wait` 或者 `wait_with_output` ，除非你显式地调用 `kill` 。因为如果父进程不 `wait` 它的话，它将会变成一个僵尸进程！！！

注： 以上问题为Linux下Python多进程的日常问题，已经见怪不怪了。

目录操作:简单grep

上一节我们实现了通过 `Command` 调用subprocess。这一节，我们将通过自己的代码去实现一个简单的grep。当然了，这种基础的工具你是能找到源码的，而我们的实现也并不像真正的grep那样注重效率，本节的主要作用就在于演示标准库API的使用。

首先，我们需要对当前目录进行递归，遍历，每当查找到文件的时候，我们回调一个函数。

于是，我们就有了这么个函数：

```
use std::env::args;
use std::io;
use std::fs::{self, File, DirEntry};
use std::path::Path;

fn visit_dirs(dir: &Path, pattern: &String, cb: &Fn(&DirEntry, &
String)) -> io::Result<()> {
    if try!(fs::metadata(dir)).is_dir() {
        for entry in try!(fs::read_dir(dir)) {
            let entry = try!(entry);
            if try!(fs::metadata(entry.path())).is_dir() {
                try!(visit_dirs(&entry.path(), pattern, cb));
            } else {
                cb(&entry, pattern);
            }
        }
    }else{
        let entry = try!(try!(fs::read_dir(dir)).next().unwrap()
);
        cb(&entry, pattern);
    }
    Ok(())
}
```

我们有了这样的一个函数，有同学可能觉得这代码眼熟。这不是标准库里的例子改了一下么？

.

.

.

是啊！

好了，继续，我们需要读取每个查到的文件，同时判断每一行里有没有所查找的内容。 我们用一个BufferIO去读取各个文件，同时用String的自带方法来判断内容是否存在。

```
fn call_back(de: &DirEntry, pt: &String) {
    let mut f = File::open(de.path()).unwrap();
    let mut buf = io::BufReader::new(f);
    for line in io::BufRead::lines(buf) {
        let line = line.unwrap_or("".to_string());
        if line.contains(pt) {
            println!("{}", &line);
        }
    }
}
```

最后，我们将整个函数调用起来，如下：

```
use std::env::args;
use std::io;
use std::fs::{self, File, DirEntry};
use std::path::Path;

fn visit_dirs(dir: &Path, pattern: &String, cb: &Fn(&DirEntry, &
String)) -> io::Result<()> {
    if try!(fs::metadata(dir)).is_dir() {
        for entry in try!(fs::read_dir(dir)) {
            let entry = try!(entry);
            if try!(fs::metadata(entry.path())).is_dir() {
                try!(visit_dirs(&entry.path(), pattern, cb));
```

```
            } else {
                cb(&entry, pattern);
            }
        }
    }else{
        let entry = try!(try!(fs::read_dir(dir)).next().unwrap()
);
        cb(&entry, pattern);
    }
    Ok(())
}

fn call_back(de: &DirEntry, pt: &String) {
    let mut f = File::open(de.path()).unwrap();
    let mut buf = io::BufReader::new(f);
    for line in io::BufRead::lines(buf) {
        let line = line.unwrap_or("".to_string());
        if line.contains(pt) {
            println!("{}", &line);
        }
    }
}

// 实现调用grep命令搜索文件
fn main() {
    let mut arg_iter = args();
    arg_iter.next();
    // panic if there is no one
    let pattern = arg_iter.next().unwrap_or("main".to_string());
    let pt =  arg_iter.next().unwrap_or("./".to_string());
    let pt = Path::new(&pt);
    visit_dirs(&pt, &pattern, &call_back).unwrap();
}
```

调用如下：

```
➜  demo git:(master) ✗ ./target/debug/demo "fn main()" ../
fn main() {
fn main() { }
fn main() {
    pub fn main() {
    pub fn main() {}
fn main() {
    pub fn main() {
    pub fn main() {}
```

网络模块:W猫的回音

本例子中，W猫将带大家写一个大家都写过但是没什么人用过的TCP ECHO软件，作为本章的结尾。本程序仅作为实例程序，我个人估计也没有人在实际的生活中去使用她。不过，作为标准库的示例来说，已经足够。

首先，我们需要一个一个服务器端。

```
fn server<A: ToSocketAddrs>(addr: A) -> io::Result<()> {
    // 建立一个监听程序
    let listener = try!(TcpListener::bind(&addr)) ;
    // 这个程序一次只需处理一个链接就好
    for stream in listener.incoming() {
        // 通过match再次解包 stream到
        match stream {
            // 这里匹配的重点是如何将一个mut的匹配传给一个Result
            Ok(mut st) => {
                // 我们总是要求client端先发送数据
                // 准备一个超大的缓冲区
                // 当然了，在实际的生活中我们一般会采用环形缓冲来重复利用
内存。
                // 这里仅作演示，是一种很低效的做法
                let mut buf: Vec<u8> = vec![0u8; 1024];
                // 通过try!方法来解包
                // try!方法的重点是需要有特定的Error类型与之配合
                let rcount = try!(st.read(&mut buf));
                // 只输出缓冲区里读取到的内容
                println!("{:?}", &buf[0..rcount]);
                // 回写内容
                let wcount = try!(st.write(&buf[0..rcount]));
                // 以下代码实际上算是逻辑处理
                // 并非标准库的一部分了
                if rcount != wcount {
                    panic!("Not Fully Echo!, r={}, w={}", rcount
, wcount);
                }
                // 清除掉已经读到的内容
                buf.clear();
```

```
            }
            Err(e) => {
                panic!("{}", e);
            }
        }
    }
    // 关闭掉Serve端的链接
    drop(listener);
    Ok(())
}
```

然后，我们准备一个模拟TCP短链接的客户端：

```
fn client<A: ToSocketAddrs>(addr: A) -> io::Result<()> {

    let mut buf = vec![0u8;1024];
    loop {
        // 对比Listener，TcpStream就简单很多了
        // 本次模拟的是tcp短链接的过程，可以看作是一个典型的HTTP交互的基
础IO模拟
        // 当然，这个通讯里面并没有HTTP协议 XD！
        let mut stream = TcpStream::connect(&addr).unwrap();
        let msg = "WaySLOG comming!".as_bytes();
        // 避免发送数据太快而刷屏
        thread::sleep_ms(100);
        let rcount = try!(stream.write(&msg));
        let _ = try!(stream.read(&mut buf));
        println!("{:?}", &buf[0..rcount]);
        buf.clear();
    }
    Ok(())
}
```

将我们的程序拼接起来如下：

```
use std::net::*;
use std::io;
use std::io::{Read, Write};
use std::env;
use std::thread;

fn server<A: ToSocketAddrs>(addr: A) -> io::Result<()> { .. }


fn client<A: ToSocketAddrs>(addr: A) -> io::Result<()> { .. }


fn main() {
    let mut args = env::args();
    args.next();
    let action = args.next().unwrap();
    if action == "s" {
        server(&args.next().unwrap()).unwrap();
    } else {
        client(&args.next().unwrap()).unwrap();
    }
}
```

各位可以自己试一下结果

写网络程序，注定了要处理各种神奇的条件和错误，定义自己的数据结构，粘包问题等都是需要我们去处理和关注的。相较而言，Rust本身在网络方面的基础设施建设并不尽如人意，甚至连网络I/O都只提供了如上的block I/O 。可能其团队更关注于语言基础语法特性和编译的改进，但其实，有着官方出品的这种网络库是非常重要的。同时，我也希望Rust能够涌现出更多的网络库方案，让Rust的明天更好更光明。

实战篇

本章举 3 个实际中的例子，来小小展示一下 Rust 在实际中的应用。它们分别是：

- Json处理
- Web 应用开发入门
- 使用Postgresql数据库

Rust json处理

JSON是一种比较重要的格式，尤其是现在的web开发领域，JSON相比于传统的XML更加容易操作和减小传输。

Rust中的JSON处理依赖 cargo 中的rustc-serialize模块

先简单的创建一个Rust项目工程

```
$ cargo new json_data --bin
```

生成文件树：

```
vagrant@ubuntu-14:~/tmp/test/rustprimer$ tree
.
`-- json_data
    |-- Cargo.toml
    `-- src
        `-- main.rs
```

生成项目 `json_data`,项目下文件介绍：

- Caogo.toml ，文件中填写一些项目的相关信息，比如版本号，联系人，项目名，文件的内容如下：

```
[package]
name = "json_data"
version = "0.1.0"
authors = ["wangxxx <xxxxx@qq.com>"]

[dependencies]
```

- src 中放置项目的源代码，main.rs 为项目的入口文件。

一些必要的了解

rustc-serialize 这个是第三方的模块，需要从cargo下载。 下载很简单，只需修改一下cargo.toml文件就行了.

```
[package]
name = "json_data"
version = "0.1.0"
authors = ["wangxxx <xxxxx@qq.com>"]

[dependencies]
rustc-serialize = "0.3.18"
```

然后执行在当前目录执行:

```
$ cargo build
```

注意一个问题由于国内网络访问*github*不稳定，这些第三方库很多托管在*github*上，所以可能需要修改你的 网络访问

1. 在安装Rust之后，会在你的用户目录之下生成一个 `.cargo` 文件夹，进入这个文件夹
2. 在 `.cargo` 文件夹下，创建一个 `config` 文件，在文件中填写中科大软件源，可能以后会出现其他的源，先用这个
3. `config` 文件内容如下

```
[registry]
index = "git://crates.mirrors.ustc.edu.cn/index"
```

cargo build 执行之后的提示信息

```
   Updating registry `git://crates.mirrors.ustc.edu.cn/index`
 Downloading rustc-serialize v0.3.18 (registry git://crates.mirr
ors.ustc.edu.cn/index)
   Compiling rustc-serialize v0.3.18 (registry git://crates.mirr
ors.ustc.edu.cn/index)
   Compiling json_data v0.1.0 (file:///home/vagrant/tmp/test/rus
tprimer/json_data)
```

再次执行tree命令:

```
.
|-- Cargo.lock
|-- Cargo.toml
|-- src
|   `-- main.rs
`-- target
    `-- debug
        |-- build
        |-- deps
        |   `-- librustc_serialize-d27006e102b906b6.rlib
        |-- examples
        |-- json_data
        `-- native
```

可以看到多了很多文件，重点关注 `cargo.lock`,开打文件:

```
[root]
name = "json_data"
version = "0.1.0"
dependencies = [
 "rustc-serialize 0.3.18 (registry+git://crates.mirrors.ustc.edu
.cn/index)",
]

[[package]]
name = "rustc-serialize"
version = "0.3.18"
source = "registry+git://crates.mirrors.ustc.edu.cn/index"
```

是关于项目编译的一些依赖信息

还有生成了target文件夹，生成了可执行文件json_data,因为main.rs中的执行结果就是打印 `hello world`

```
$ cargo run

Hello, world!
```

开始写代码

直接使用官方的 rustc_serialize 中的例子：

```
extern crate rustc_serialize;
// 引入rustc_serialize模块
use rustc_serialize::json;

// Automatically generate `RustcDecodable` and `RustcEncodable` trait
// implementations
// 定义TestStruct
#[derive(RustcDecodable, RustcEncodable)]
pub struct TestStruct  {
    data_int: u8,
    data_str: String,
    data_vector: Vec<u8>,
}

fn main() {
    // 初始化TestStruct
    let object = TestStruct {
        data_int: 1,
        data_str: "homura".to_string(),
        data_vector: vec![2,3,4,5],
    };

    // Serialize using `json::encode`
    // 将TestStruct转意为字符串
    let encoded = json::encode(&object).unwrap();
    println!("{}",encoded);
    // Deserialize using `json::decode`
    // 将json字符串中的数据转化成TestStruct对应的数据，相当于初始化
    let decoded: TestStruct = json::decode(&encoded).unwrap();
    println!("{:?}",decoded.data_vector);
}
```

当然我们也可以在文本中作为api的返回结果使用，下来的章节中，我们将讨论这个问题

rust web 开发

rust既然是系统级的编程语言，所以当然也能用来开发 web,不过想我这样凡夫俗子，肯定不能从头自己写一个 web 服务器，肯定要依赖已经存在的 rust web开发框架来完成 web 开发。

rust目前比较有名的框架是iron和nickel，我们两个都写一下简单的使用教程。

iron

接上一篇，使用cargo获取第三方库。`cargo new mysite --bin`

在cargo.toml中添加iron的依赖，

```
[dependencies]
iron = "*"
```

然后build将依赖下载到本地 `cargo build`

如果报ssl错误，那可能你需要安装linux的ssl开发库。

首先还是从 hello world 开始吧,继续抄袭官方的例子：

```
extern crate iron;

use iron::prelude::*;
use iron::status;

fn main() {
    Iron::new(|_: &mut Request| {
        Ok(Response::with((status::Ok, "Hello World!")))
    }).http("localhost:3000").unwrap();
}
```

然后运行

`cargo run`

使用curl直接就可以访问你的网站了。

`curl localhost:3000`

`Hello World!`

仔细一看，发现这个例子很无厘头啊，对于习惯了写python的我来说，确实不习惯。 简单点看：

`iron::new().http("localhost:3000").unwrap()` 这句是服务器的基本的定义，new内部是一个rust lambda 表达式

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

具体的怎么使用 ，可以暂时不用理会，因为你只要知道如何完成web，因为我也不会。。 结合之前一章节的json处理，我们来看看web接口怎么返回json,当然也要rustc_serialize 放到 cargo.toml 中

下面的代码直接参考开源代码地址

```
extern crate iron;
extern crate rustc_serialize;

use iron::prelude::*;
use iron::status;
use rustc_serialize::json;

#[derive(RustcEncodable)]
struct Greeting {
    msg: String
}

fn main() {
    fn hello_world(_: &mut Request) -> IronResult<Response> {
        let greeting = Greeting { msg: "Hello, World".to_string(
) };
        let payload = json::encode(&greeting).unwrap();
        Ok(Response::with((status::Ok, payload)))
    }

    Iron::new(hello_world).http("localhost:3000").unwrap();
    println!("On 3000");
}
```

执行 cargo run 使用 curl 测试结果:

```
curl localhost:3000
{"msg":"Hello, World"}
```

当然可以可以实现更多的业务需求，通过控制自己的json。

既然有了json了，如果要多个路由什么的，岂不是完蛋了，所以不可能这样的，我们需要考虑一下怎么实现路由的定制

不说话直接上代码，同一样要在你的cargo.toml文件中添加对router的依赖

```
extern crate iron;
extern crate router;
```

```
extern crate rustc_serialize;

use iron::prelude::*;
use iron::status;
use router::Router;
use rustc_serialize::json;

#[derive(RustcEncodable, RustcDecodable)]
struct Greeting {
    msg: String
}

fn main() {
    let mut router = Router::new();

    router.get("/", hello_world);
    router.post("/set", set_greeting);

    fn hello_world(_: &mut Request) -> IronResult<Response> {
        let greeting = Greeting { msg: "Hello, World".to_string(
) };
        let payload = json::encode(&greeting).unwrap();
        Ok(Response::with((status::Ok, payload)))
    }

    // Receive a message by POST and play it back.
    fn set_greeting(request: &mut Request) -> IronResult<Respons
e> {
        let payload = request.body.read_to_string();
        let request: Greeting = json::decode(payload).unwrap();
        let greeting = Greeting { msg: request.msg };
        let payload = json::encode(&greeting).unwrap();
        Ok(Response::with((status::Ok, payload)))
    }

    Iron::new(router).http("localhost:3000").unwrap();
}
```

这次添加了路由的实现和获取客户端发送过来的数据，有了get，post,所以现在一个基本的api网站已经完成了。不过 并不是所有的网站都是api来访问，同样需要html模版引擎和直接返回静态页面。等等

```
vagrant@ubuntu-14:~/tmp/test/rustprimer/mysite$ cargo build
   Compiling mysite v0.1.0 (file:///home/vagrant/tmp/test/rustpr
imer/mysite)
src/main.rs:29:36: 29:52 error: no method named `read_to_string`
 found for type `iron::request::Body<'_, '_>` in the current sco
pe
src/main.rs:29         let payload = request.body.read_to_string
();
                                                  ^~~~~~~~~~~~~~
~~
src/main.rs:29:36: 29:52 help: items from traits can only be use
d if the trait is in scope; the following trait is implemented b
ut not in scope, perhaps add a `use` for it:
src/main.rs:29:36: 29:52 help: candidate #1: use `std::io::Read`
error: aborting due to previous error
Could not compile `mysite`.
```

编译出错了，太糟糕了，提示说没有read_to_string这个方法，然后我去文档查了一下，发现有read_to_string方法 再看提示信息

```
src/main.rs:29:36: 29:52 help: items from traits can only be use
d if the trait is in scope; the following trait is implemented b
ut not in scope, perhaps add a `use` for it:
src/main.rs:29:36: 29:52 help: candidate #1: use `std::io::Read`
```

让我们添加一个 `std::io::Read` ,这个如果操作过文件，你一定知道怎么写，添加一下，应该能过去了，还是继续出错了，看看报错

```
   Compiling mysite v0.1.0 (file:///home/vagrant/tmp/test/rustpr
imer/mysite)
src/main.rs:30:36: 30:52 error: this function takes 1 parameter
but 0 parameters were supplied [E0061]
src/main.rs:30         let payload = request.body.read_to_string
();
```

```
                                                    ^~~~~~~~~~~~~~
~~
src/main.rs:30:36: 30:52 help: run `rustc --explain E0061` to se
e a detailed explanation
src/main.rs:31:46: 31:53 error: mismatched types:
 expected `&str`,
    found `core::result::Result<usize, std::io::error::Error>`
(expected &-ptr,
    found enum `core::result::Result`) [E0308]
src/main.rs:31         let request: Greeting = json::decode(payl
oad).unwrap();
                                                            ^~~~
~~~
src/main.rs:31:46: 31:53 help: run `rustc --explain E0308` to se
e a detailed explanation
src/main.rs:30:36: 30:52 error: cannot infer an appropriate life
time for lifetime parameter `'b` due to conflicting requirements
 [E0495]
src/main.rs:30         let payload = request.body.read_to_string
();
                                                    ^~~~~~~~~~~~~~
~~
src/main.rs:29:5: 35:6 help: consider using an explicit lifetime
 parameter as shown: fn set_greeting<'a>(request: &mut Request<'
a, 'a>) -> IronResult<Response>
src/main.rs:29     fn set_greeting(request: &mut Request) -> Iro
nResult<Response> {
src/main.rs:30         let payload = request.body.read_to_string
();
src/main.rs:31         let request: Greeting = json::decode(payl
oad).unwrap();
src/main.rs:32         let greeting = Greeting { msg: request.ms
g };
src/main.rs:33         let payload = json::encode(&greeting).unw
rap();
src/main.rs:34         Ok(Response::with((status::Ok, payload)))
               ...
error: aborting due to 3 previous errors
Could not compile `mysite`.
```

第一句提示我们，这个read_to_string(),至少要有一个参数，但是我们一个都没有提供。 我们看看read_to_string的用法

```
se std::io;
use std::io::prelude::*;
use std::fs::File;

let mut f = try!(File::open("foo.txt"));
let mut buffer = String::new();

try!(f.read_to_string(&mut buffer));
```

用法比较简单，我们修改一下刚刚的函数：

```
fn set_greeting(request: &mut Request) -> IronResult<Response> {
        let mut payload = String::new();
        request.body.read_to_string(&mut payload);
        let request: Greeting = json::decode(&payload).unwrap();
        let greeting = Greeting { msg: request.msg };
        let payload = json::encode(&greeting).unwrap();
        Ok(Response::with((status::Ok, payload)))
    }
```

从request中读取字符串，读取的结果存放到payload中，然后就可以进行操作了，编译之后运行，使用curl提交一个post数据

```
$curl -X POST -d '{"msg":"Just trust the Rust"}' http://localhos
t:3000/set
{"msg":"Just trust the Rust"}
```

iron 基本告一段落 当然还有如何使用html模版引擎，那就是直接看文档就行了。

nickel

当然既然是web框架肯定是iron能干的nicke也能干，所以那我们就看看如何做一个hello 和返回一个html 的页面

同样我们创建 `cargo new site --bin` ，然后添加nickel到cargo.toml中，`cargo build`

```
#[macro_use] extern crate nickel;

use nickel::Nickel;

fn main() {
    let mut server = Nickel::new();

    server.utilize(router! {
        get "**" => |_req, _res| {
            "Hello world!"
        }
    });

    server.listen("127.0.0.1:6767");
}
```

简单来看，也就是这样回事。

1. 引入了nickel的宏
2. 初始化Nickel
3. 调用utilize来定义路由模块。
4. `router!` 宏，传入的参数是 get 方法和对应的路径，"**"是全路径匹配。
5. listen启动服务器

当然我们要引入关于html模版相关的信息

```
#[macro_use] extern crate nickel;

use std::collections::HashMap;
use nickel::{Nickel, HttpRouter};

fn main() {
    let mut server = Nickel::new();

    server.get("/", middleware! { |_, response|
        let mut data = HashMap::new();
        data.insert("name", "user");
        return response.render("site/assets/template.tpl", &data
);
    });

    server.listen("127.0.0.1:6767");
}
```

上面的信息你可以编译，使用curl看看发现出现

```
$ curl http://127.0.0.1:6767
Internal Server Error
```

看看文档，没发现什么问题，我紧紧更换了一个文件夹的名字，这个文件夹我也创建了。 然后我在想难道是服务器将目录写死了吗？于是将上面的路径改正这个，问题解决。

```
return response.render("examples/assets/template.tpl", &data);
```

我们看一下目录结构

```
.
|-- Cargo.lock
|-- Cargo.toml
|-- examples
|   `-- assets
|       `-- template.tpl
|-- src
|   `-- main.rs
```

rust数据库操作

编程时，我们依赖数据库来存储相应的数据，很多编程语言都支持对数据库的操作，所以当然可以使用Rust操作数据库。

不过在我自己操作时，发现很多问题，主要因为我不了解Rust在操作数据库时，应该注意的事情，从而浪费了很多的时间，在进行数据查询时。 具体遇到的坑，我会做一些演示，从而让大家避免这些情况。

首先使用Rust操作PostgreSQL,因为PostgreSQL是我最喜欢的数据库。

首先创建新项目 `cargo new db --bin`

在cargo.toml中添加 `postgres` 如下：

```
[package]
name = "db"
version = "0.1.0"
authors = ["vagrant"]

[dependencies]
postgres="*"
```

当然我们还是进行最简单的操作，直接粘贴复制，代码来源

```

extern crate postgres;

use postgres::{Connection, SslMode};

struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>
}

fn main() {
    let conn = Connection::connect("postgres://postgres@localhos
```

```
t", SslMode::None)
            .unwrap();

    conn.execute("CREATE TABLE person (
                    id              SERIAL PRIMARY KEY,
                    name            VARCHAR NOT NULL,
                    data            BYTEA
                  )", &[]).unwrap();
    let me = Person {
        id: 0,
        name: "Steven".to_string(),
        data: None
    };
    conn.execute("INSERT INTO person (name, data) VALUES ($1, $2
)",
                 &[&me.name, &me.data]).unwrap();

    for row in &conn.query("SELECT id, name, data FROM person",
&[]).unwrap() {
        let person = Person {
            id: row.get(0),
            name: row.get(1),
            data: row.get(2)
        };
        println!("Found person {}", person.name);
    }
}
```

这些简单的，当然不是我们想要的东西，我们想要的是能够进行一些分层，也就是基本的一些函数逻辑划分，而不是在一个main函数中，完成所有的一切。

创建lib.rs文件

从上到下来看文件：

1. 首先导入postgres的各种库
2. 创建一个Person 的struct，按照需求的字段和类型。
3. 创建一个连接函数，返回连接对象。
4. 创建一个插入函数，用来插入数据

5. 创建一个查询函数，用来查询数据
6. 创建一个查询函数，用来查询所有的数据。

当然这些函数都是有一定的功能局限性。

```
extern crate postgres;

use postgres::{Connection, SslMode};
use postgres::types::FromSql;
use postgres::Result as PgResult;


struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>
}


pub fn connect() -> Connection{
    let dsn = "postgresql://postgres:2015@localhost/rust_example"
;
    Connection::connect(dsn, SslMode::None).unwrap()
}

pub fn insert_info(conn : &Connection,title : &str, body: &str){

    let stmt = match conn.prepare("insert into blog (title, body
) values ($1, $2)") {
        Ok(stmt) => stmt,
        Err(e) => {
            println!("Preparing query failed: {:?}", e);
            return;
        }
    };
        stmt.execute(&[&title, &body]).expect("Inserting blogpos
ts failed");
}
```

```
pub fn query<T>(conn: &Connection,query: &str) ->PgResult<T>
        where T: FromSql {
            println!("Executing query: {}", query);
            let stmt = try!(conn.prepare(query));
            let rows = try!(stmt.query(&[]));
            &rows.iter().next().unwrap();
            let row = &rows.iter().next().unwrap();
                //rows.iter().next().unwrap()
            row.get_opt(2).unwrap()

}

pub fn query_all(conn: &Connection,query: &str){
            println!("Executing query: {}", query);
            for row in &conn.query(query,&[]).unwrap(){
                let person = Person{
                    id: row.get(0),
                    name: row.get(1),
                    data: row.get(2)
            };
            println!("Found person {}", person.name);
            }

}
```

然后在main.rs 中调用相应的函数代码如下

1. extern db ,引入db，也就是将项目本身引入
2. use db 使用db，中的可以被引入的函数
3. 定义Blog,由于个人使用blog表，是自己创建，所以如果报错说不存在表，需要你自己去创建
4. 使用lib中定义的函数，进行最基本的一些操作

```
extern crate postgres;
extern crate db;

use postgres::{Connection, SslMode};

use db::*;

struct Blog {
    title: String,
    body:  String,
}

fn main() {
    let conn:Connection=connect();

    let blog = Blog{
        title: String::from("title"),
        body: String::from("body"),
    };
    let title = blog.title.to_string();
    let body = blog.body.to_string();
    insert_info(&conn,&title,&body);

   for row in query::<String>(&conn,"select * from blog"){
        println!("{:?}",row);
    }
    let sql = "select * from person";
    query_all(&conn,&sql);
}
```

自己遇到的坑

- 创建连接函数时，连接必须有一个返回值，所以必须指定返回值的类型， 对于一个写Python的人而言，我觉得是痛苦的，我想按照官方的写法match 一下，发现可能产生多个返回值。在编译时直接无法通过编译，所以最终 使用了unwrap,解决问题，不过我还是没有学会，函数多值返回时我如何 定义返回值

- 在使用 `&conn.query(query,&[]).unwrap()` 时，我按照文档操作，文档说返回的是一个可迭代的数据，那也就是说，我可以使用for循环，将数据打印，但是发现怎么也不能实现：

```
pub fn query_all(conn: &Connection,query: &str){
            println!("Executing query: {}", query);
            for row in &conn.query(query,&[]).unwrap(){
                  println!("Found person {:?}", row.get_opt(1));
            }
}
```

报错如下：

```
vagrant@ubuntu-14:~/tmp/test/rustprimer/db$ cargo run
   Compiling db v0.1.0 (file:///home/vagrant/tmp/test/rustprimer
/db)
src/lib.rs:53:37: 53:47 error: unable to infer enough type infor
mation about `_`; type annotations or generic parameter binding
required [E0282]
src/lib.rs:53    println!("Found person {:?}", row.get_opt(1));
                                                  ^~~~~~~~~~
<std macros>:2:25: 2:56 note: in this expansion of format_args!
<std macros>:3:1: 3:54 note: in this expansion of print! (define
d in <std macros>)
src/lib.rs:53:3: 53:49 note: in this expansion of println! (defi
ned in <std macros>)
src/lib.rs:53:37: 53:47 help: run `rustc --explain E0282` to see
 a detailed explanation
error: aborting due to previous error
Could not compile `db`.
```

然后去查看了关于postgres模块的所有函数，尝试了无数种办法，依旧没有解决。

可能自己眼高手低，如果从头再把Rust的相关教程看一下，可能很早就发现这个问题， 也有可能是因为习惯了写Python，导致自己使用固有的思维来看待问题和钻牛角尖，才 导致出现这样的问题，浪费很多的时间。

- 改变思维，把自己当作一个全新的新手，既要利用已有的思想来学习新的语

言，同样不要 被自己很精通的语言，固化自己的思维。

附录I-术语表

- ADT(Algebraic Data Type:代数数据类型):
- ARC(Atomic Reference Counting:原子引用计数):
- associated function(关联函数):
- associated type(关联类型): Trait 里面可以有关联类型
- AST(Abstract Syntax Tree:抽象语法树):
- benchmark(基准测试):
- bitwise copy:
- borrow(借用):
- bounds(约束):
- box:
- byte string():
- cargo:
- cast:
- channel:
- coercion:
- constructor(构造器):
- consumer:
- copy:
- crate:
- dangling pointer:
- deref(解引用):
- derive:
- designator(指示符):
- destructor():
- destructure(析构):
- diverging function(发散函数):
- drop:
- DST(Dynamically Sized Type):
- dynamic dispatch(动态分发):
- enum():
- feature gate(特性开关): nightly 版本中有特性开关可以启用一些实验性质的特性

- FFI(Foreign Function Interface:外部函数接口):
- guard:
- hygiene:
- inline function(内联函数):
- item:
- iterator(迭代器):
- iterator adaptor(迭代器适配器):
- lifetime(生命周期):
- lifetime elision:
- literal string():
- macro by example:
- memberwise copy:
- module(模块)
- move:
- option:
- ownership(所有权):
- panic(崩溃):
- phantom type:
- primitive type(基本类型): 整型、浮点、布尔等基本类型
- procedural macro:
- RAII():
- raw string:
- raw pointer:
- RC(Reference Counting:引用计数)
- result:
- shadowing:
- static dispatch(静态分发):
- slice(切片): 某种数据类型的视图，例如 string, vector
- statement(): 与 expression 相区别
- trait:
- trait object:
- tuple(元组):
- UFCS(Universal Function Call Syntax)
- unit():
- unwind:
- unwrap():

- wrap: